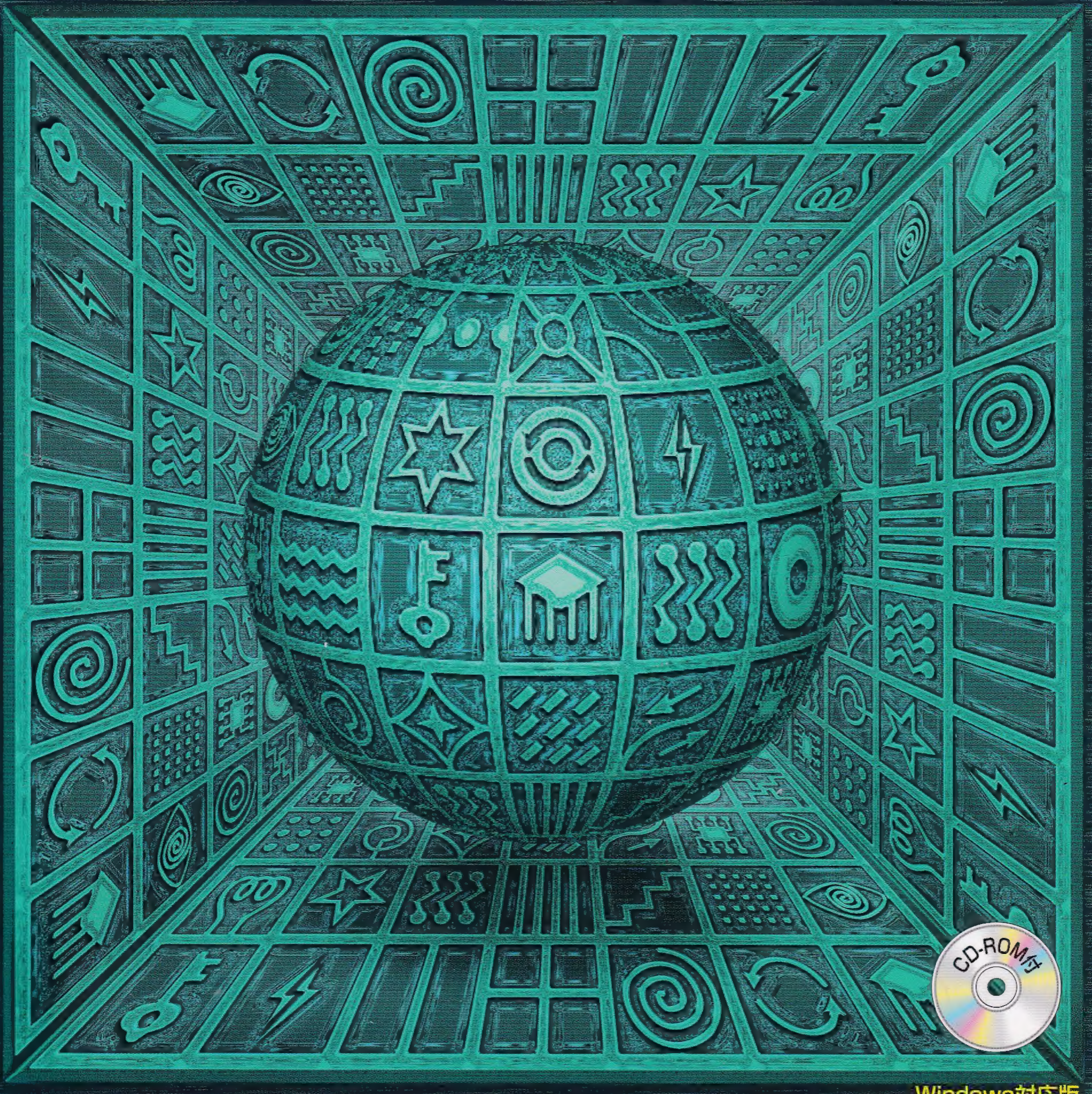
Design Wave BOOKS

改訂 ARMプロセッサ

32ビットRISCのシステム・アーキテクチャ

Steve Furber 著／アーム(株) 監訳



Windows対応版

改訂 ARMプロセッサ

32ビットRISCのシステム・アーキテクチャ

Steve Furber 著 / アーム(株) 監訳



CQ出版社

ARM System-on-chip Architecture (second edition) by Steve Furber

日本語翻訳では，アーム株式会社の協力を得ました．
本書付録Dおよび付属CD-ROMについては，
Copyright © 1999-2000 by ARM Limited
です．これらの収録に関してアーム株式会社の協力を得ました．

はじめに

■ねらい

本書では，マイクロプロセッサ設計と，マイクロプロセッサを中心としたシステム・オンチップ(SoC)設計とマイクロプロセッサ・コアそのものの設計に用いられる概念と方法論を紹介します．マイクロプロセッサ設計の原理については，ARMを例として詳しく図解しています．

本書のねらいは，SoCとマイクロプロセッサがどのように設計され，使用されているか，また，現代のプロセッサがなぜそのように設計されているのかを，よりよく理解してもらうことにあります．一般的原理だけを知りたい読者には，ARMの図解がつかみどころのない理論に具体性を与えることでしょう．ARMの設計をとくに知りたい読者には，一般的原理を知ることによってARMがいまのARMである理論的根拠がつかめるでしょう．

他のマイクロプロセッサ・アーキテクチャについては，本書では扱いません．アーキテクチャの比較研究をしたい読者にとっては，本書でARMについての必要情報は得られますが，他の設計については別の資料を探す必要があります．

■対象

本書は，おもに二つのグループの読者を想定しています．

(1) プロのハードウェアおよびソフトウェア技術者で，ARMプロセッサを組み込んだSoC製品の設計に従事する方，または製品向けにARMを評価している方にとって，本書は役に立つでしょう．ARMの技術資料と重なる部分も多くありますが，本書ではより幅広い関連と背景が述べられています．本書では省略した箇所もあるため，メーカが供給するデータシートの代用にはなりません．しかし，基本的な全体像をつかみ，データを補足するのに有効です．

(2) コンピュータ・サイエンス，コンピュータ工学，電気工学の学生にとっては，学習段階の所々で本書が役に立つでしょう．いくつかの章は，以前，学部学生を教えるのに使った教材に基づいています．また，内容の一部は大学院課程から引用しています．

■必要な知識

本書は，コンピュータ・アーキテクチャやコンピュータ論理設計の入門書ではありません．読者は，これらの分野に関してコンピュータ・サイエンスまたはコンピュータ工学の学部2年生程度の知識をもっていることが前提となっています．1年生程度の内容も出てきますが，最初の導入部分としてよりも記憶の再確認を目的としています．

ARMプロセッサに関する知識はまったく必要ありません．

■ARMについて

1985年4月26日，初代ARMのプロトタイプが英国ケンブリッジ市のAcorn Computers Limitedに到着しました．米国カリフォルニア州サンノゼ市のVLSI Technology, Inc.で製造されたものです．数時間後，プロトタイプはコードを処理し始め，お祝いのMoët&Chandoa(シャンペン)の栓が抜かれました．その後1980年代が終わるまで，ARMは英国のコンピュータ教育の基盤となるAcornのデスクトップ製品を支えるため静かに開発が続けられました．

そして1990年代，ARM LimitedのもとでARMは世界の舞台へと飛躍し，高性能低コスト組み込み型アプリケーションの分野で市場をリードする地位を確立したのです．市場における卓越した地位のおかげで，ARMのリソースは増え，新しいARMベースの開発も次々に始まりました．

ここ10年間のハイライトは次のとおりです．

(1) 小型システムにおいてコストと消費電力を低減する「Thumb（サム）」と呼ばれる新しい圧縮命令形式の登場
(2) 「ARM9」「ARM10」「StrongARM」の各プロセッサ・ファミリによる性能の飛躍的向上
(3) 最先端ソフトウェア開発・デバッグ環境
(4) ARMプロセッサ・コアをベースとした非常に広範な組み込み型アプリケーション

現代のSoCおよびプロセッサ設計の基本はほとんど，ARMファミリのどこかに現れてきます．また，ARMによって導入への道が開かれた概念（命令ストリームの動的伸張など）もあります．複雑なシステムチップの奥深く組み込まれるARMコアを使ったシステムのデバッグは今日の最先端の開発技術を表している一方で，3段パイプライン構成のARMコアの簡潔さのおかげで，ARMコアは実際のプロセッサ設計のための教育的な導入例としても適しています．

■本書の構成

第1章は，大学1年生レベルのプロセッサ設計を思い出すところから始まります．論理とゲート・レベル表現の役割を復習しながらハードウェアの抽象化の原則を説明します．次に，後出する内容に備えて「縮小命令セット・コンピュータ」（RISC）という重要な概念を紹介し，低消費電力設計について少し触れて章を締めくくります．第2章では，第1章で紹介した概念に沿ってARMプロセッサ・アーキテクチャを説明します．

第3章ではユーザ・レベルのアセンブリ言語プログラミングをやさしく紹介しているため，大学1年生の授業で取り上げてもよいでしょう．

第4章では，3段および5段パイプラインARMプロセッサ・コアの構成と実装について大学2年生を教えるのに適したレベルで説明するとともに，実装上の問題についても触れます．

第5章，第6章では，ARM命令セット・アーキテクチャの奥深くに入っていきます．第5章では，各命令のバイナリ表記など，命令セットについて第3章より詳しく，かつ細かい部分にまで触れます．一読したあと，必要に応じて参照してもらうのがよいでしょう．第6章は，命令セットから少し離れて，高級言語（ここではC）に本当に必要なものは何か，またARM命令セットがその必要性をどのように満たしているかを考えます．この章は大学2年生の教材に基づいています．

第7章では「Thumb（サム）」命令セット，つまり小型組み込み型システムのコード密度と消費電力におけるARMの革新的技術を紹介します．コンピュータ科学の一般的な研究にとっては末節的な問題ですが，大学院レベルにとっては興味深い別の視点を提供してくれます．

第8章は，組み込み型プロセッサ・コアを使用するシステムのデバッグと基板レベル・システムのプロダクション・テストにおける問題を提起します．これらの問題は第9章の背景となります．

第9章では，さまざまなARM整数コアを紹介し，第4章のテーマを広げて「Thumb」，デバッグ・ハードウェア，より洗練されたパイプライン動作，完全な非同期的型動作をもつコアにも触れます．

第10章では，メモリ階層の概念を紹介し，メモリ管理とキャッシュの原理を考えます．

第11章では，大学2年生レベルで，現代のOS（operating system）の必要条件を復習します．また，そのような必要条件に対応するためにARMが採用した方法について説明します．

第12章では，メモリ管理を完全にサポートする統合ARM CPUコア（StrongARMなど）を紹介します．

第13章では，組み込み型プロセッサ・コアをもつSoCの設計に関する問題に触れます．これこそ，ARMが

最先端技術を誇る分野です．組み込み型システム・チップの製造における例をあげ，複雑な特定アプリケーション向けシステムをシリコンに組み込む際に必ず直面する多くの問題への解決策を示します．

第14章では，ARMの本筋から離れ，1990年代に英国のマンチェスター大学で開発されたARM互換非同期プロセッサおよびシステムについて説明します．10年間にわたって研究されたAMULETテクノロジは，本書執筆中の現在，商業化への第一歩を踏み出そうとしています．章の最後では、32ビット非同期マイクロプロセッサ初の商用アプリケーションであるDRACO SoC設計についても述べます。

短い付録では，第1章で使用するコンピュータ論理設計の基礎と用語を紹介します．

本書の終わりには，本書で使用した用語集と，さらなる研究のための文献目録，そして詳細なインデクスを添えました．

■大学課程への対応

各大学課程レベルにふさわしい章は次のとおりです．

1年生：第1章(基本的プロセッサ設計)，第3章(アセンブリ言語プログラミング)，第5章(アセンブリ言語プログラミング用の命令フォーマットとリファレンス)

2年生：第4章(単純なパイプライン・プロセッサ設計)，第6章(高級言語のためのアーキテクチャ・サポート)，第10章，第11章(メモリ階層とOSのためのアーキテクチャ・サポート)

3年生：第8章(組み込み型システムのデバッグとテスト)，第9章(高度なパイプライン・プロセッサ設計)，第12章(高度なCPU)，第13章(組み込み型システムの例)

大学院レベルでは，複数の章にまたがるテーマを追究するのがいいかもしれません．たとえば，プロセッサ設計(第1，2，4，9，10，12章)，命令セット設計(第2，3，5，6，7，11章)，組み込み型システム(第2，4，5，8，9，13章)などです．

第14章は，非同期設計について大学課程3年生または高度な大学院レベルにふさわしい内容を含んでいます．しかし，さらに詳しい追加資料も(本書には書かれていない)大量に必要です．

■謝辞

この10年間にわたって，多くの方々がARMの開発に貢献しました．著者の主義として，各開発の主要人物名を本文中に言及しませんでした．なぜなら，人名のリストがあまりに長くなってしまい，かといって切り詰めてしまうのは不本意だからです．歴史というものは，一人か二人の目立った人物の功績としてしまう傾向がありますが，これでは謙虚な人々が影に隠れてしまいます．とはいえ，ARMについての本書を執筆するにあたって，Sophie Wilsonの存在は重要でした．彼女の命令セット・アーキテクチャの原型は今も生きていますし，拡張されてこそいますが今日までほとんど変化していません．

また，スタッフに会う機会と，設計資料を見せてくれたARM社(ARM Limited)の協力に感謝します．ARMの半導体パートナ，とくにVLSI Technology, Inc.(現在はPhilips Semiconductorsの一部門)の協力にも感謝はつきません．

本書は，草稿段階で読んでくださった方からの貴重なコメントのおかげでかなり充実しました．草稿に対する好意的な感想，改善のための直接的な意見すべてに感謝します．出版社Addison Wesley Longman Limitedにも，そのような意見に対する対応，著者としてのさまざまな面で多大な助力をいただきました．

最後に，本書執筆のために家族の時間を割いてくれた妻Valerie，娘AlisonとCatherineに謝辞を捧げます．

Steve Furber

2000年3月

目　次

第1章 プロセッサ設計とは

汎用プロセッサの設計では，多くの工学的分野でなされている努力と同じように，トレードオフと妥協点を深く考慮することが求められます．本章では，プロセッサの命令セットと論理設計の基本概念，また技術者向け設計テクニックを紹介します．

複雑なコンピュータを理解するには「抽象化」が重要です．ここでは，コンピュータ・ハードウェア設計者が用いる論理ゲートによる抽象化を紹介します．単純なプロセッサの設計を例にして，命令セットからレジスタ転送レベル(RTL)の記述，論理ゲートへと，順に説明します．

縮小命令セット・コンピュータ(RISC)の基になるアイデアは，1980年代，米国スタンフォード大学と，カリフォルニア大学バークレー校のプロセッサ研究課程で生まれました(もっとも，核となるアイデアはさらに昔のマシンに遡るが)．この章では，RISCへの流れをつくった考え方に着目し，後の章で説明するARMプロセッサの設計に影響を与えるに至った思想について考えます．

携帯用コンピュータ製品の市場が急速に拡大するにつれ，ディジタル回路の電力消費が重要な課題になってきています．本章の最後に，低電力，高性能設計の基本を紹介します．

1.1 プロセッサのアーキテクチャと構成

現在の汎用コンピュータは，すべて「プログラム内蔵方式ディジタル・コンピュータ」の原理に基づいています．プログラム内蔵方式の概念は1940年代，米国プリンストン高等研究所(Princeton Institute for Advanced Study)で考案され，1948年6月，英国マンチェスター大学で開発された“Baby”マシンにおいて初めて実装されました．

それ以後50年間にわたる開発で，プロセッサの性能は飛躍的に向上し，同時にコストも目覚ましく低減しました．コストが着実に低減したにもかかわらず，コンピュータ動作の基本は意外にもほとんど変化していません．進歩の大部分はエレクトロニクス技術の進歩に起因しています．真空管からトランジスタへ，そしてバイポーラ・トランジスタを組み込んだICへ，さらにIC技術の発展により1チップ上に何百万個ものFETを内蔵する今日のVLSIが生まれました．トランジスタが小型化するにつれ，低価格，高速，低消費電力化も進みました．この相乗効果は，ここ30年間コンピュータ業界を引っ張っていますし，少なくともあと数年間は，同じ状態が続くでしょう．

しかし，過去50年間の進歩が，すべてエレクトロニクス技術に由来するものばかりではありません．技術の応用に関する新しい発想がすばらしい進歩をもたらしたこともありました．これらの発想について説明するにあたり，「コンピュータ・アーキテクチャ」と「コンピュータ構成」の用語の解釈から述べることにします．

(1) コンピュータ・アーキテクチャ(computer architecture)

コンピュータ・アーキテクチャは，ユーザからコンピュータを見た場合，その見え方を述べたものです．命令セット，可視レジスタ，メモリ管理テーブル構造，例外処理モデルなどは，すべてアーキテクチャの一部です．

(2) コンピュータ構成(computer organization)

コンピュータ構成は，ユーザからは見えないアーキテクチャを述べたものです．パイプライン構造，トランスペアレント・キャッシュ，テーブル・ウォーキング・ハードウェア，TLB(変換ルックアサイド・バッファ)などは，このコンピュータ構成にあたります．

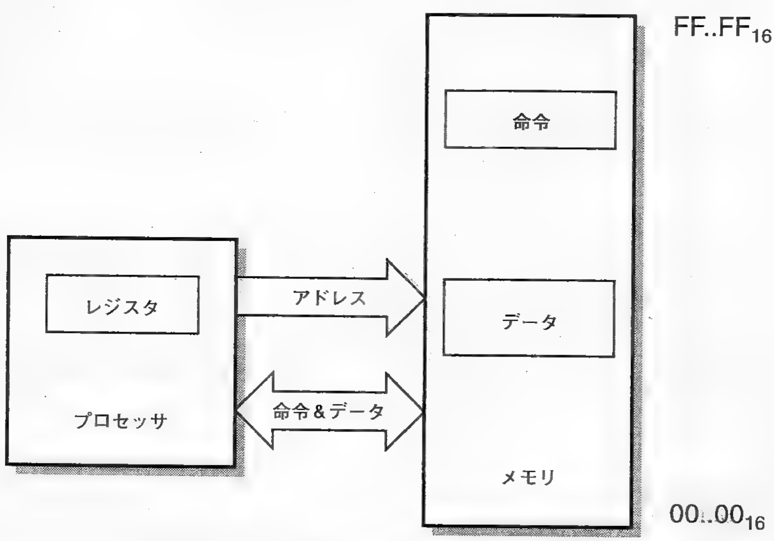
コンピュータ設計の発展史上，1960年代初期の仮想メモリ，トランスペアレント・キャッシュ，パイプライン構造などの導入は大きな節目となりました．RISCのアイデアは，コンピュータ技術の費用対性能を決定する平衡点を大幅にシフトさせたという点において，これらの進歩に匹敵します．

◆プロセッサとは何か

汎用プロセッサは，メモリ上の命令を実行する有限オートマトンです．メモリに保持した数値とプロセッサ内部のレジスタに保持した数値(図1.1参照．メモリ・アドレスの16進法表記は6.2節を参照)により，システムの状態が決定されます．各命令は全体の状態変化の詳細を定義し，次にどの命令が実行されるかも定義します．

◆プログラム内蔵方式コンピュータ

プログラム内蔵方式コンピュータは，命令とデータを同一のメモリ・システムに保存しているため，必要なら命令もデータとして扱うことができます．これにより，プロセッサ自体が次に実行する命令を生成することができます．これを細かい粒度で実行するプログラム(「自己変更」コード)はデバッグしにくいため現在では一般に望ましくないとされていますが，粗い粒度で使用することは多くのコンピュータ処理の基本となっています．ディスクから新しいプログラムをロードして実行する(旧いプログラムを上書きする)たびに，コンピュータは自己プログラムを変更する能力を使用しています．



〔図1.1〕プログラム内蔵方式コンピュータの状態

◆コンピュータ・アプリケーション

そのプログラム性のおかげで，プログラム内蔵式のコンピュータは「万能」であり，それは適当なアルゴリズムによって記述可能などんなタスクでも実行可能であることを意味します．この能力によって，ユーザが様様な状況で様々なプログラムを実行するデスクトップマシンとして使用されることもあれば，様々なアプリケーションで使用されるあらかじめ決まったプログラムをもつプロセッサとして使用されることもあります．そのようなアプリケーションはその性質上携帯電話，自動車のエンジン制御システム等のような製品に組み込まれています．

1.2 ハードウェアの抽象化

コンピュータは非常に高速で動く複雑なマシンです．現在のマイクロプロセッサは，数百万個のトランジスタからできており，各トランジスタが毎秒1億回もスイッチングしています．デスクトップ・コンピュータの画面でドキュメントをスクロールさせながら，毎秒数億個のトランジスタ・スイッチがどのようにして機能しているのかを考えてみてください．そのスイッチのどれもが厳密な設計によって決まっているのです．ランダムに無制御にスイッチしているのではありません．たった一つのエラーによって，コンピュータは使用不可能な状態に陥ってしまいます．こんな複雑なシステムを高い信頼性で動作させるためには，どう設計すればよいのでしょう？

◆トランジスタ

手がかりは，質問そのものの中に見つけることができます．ここまで，コンピュータをトランジスタの観点から説明してきましたが，トランジスタとは何でしょう？　それは原子を構成する物質が確率的にしか存在を記述できない波のように振る舞うという奇妙な，確率的な量子力学の定理の導入によってのみ理解し得る複雑な電気的特性をもつ，注意深く選別された化学物質からなる興味深い構造体です．ただし量子力学をもちださなくても，端子における電圧とトランジスタを流れる電流の等式の組として，トランジスタ全体での振る舞いを記述できます．これらの数式はデバイスの本質的な振る舞いをその物理的動作から抽象しています．

◆論理ゲート

トランジスタの動作を表す数式は，それでもまだ複雑です．トランジスタ群を図1.2のCMOSのNANDゲートのように特定の構造に結線すると，これらのトランジスタ群の動作を単純に記述することができます．

各入力線（AとB）をV_{dd}またはV_{SS}に近い電圧にすると，以下の規則により出力もやはりV_{dd}またはV_{SS}に近い値になります．

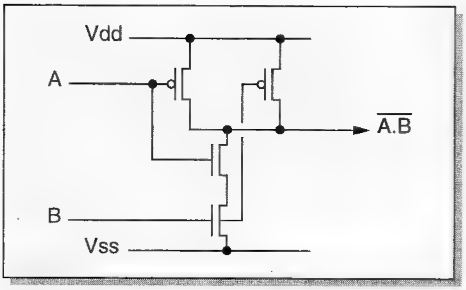
(1) AとB両方がV_{dd}に近ければ，出力はV_{SS}に近くなる

(2) AまたはB（もしくは両方）がV_{SS}に近ければ，出力はV_{dd}に近くなる

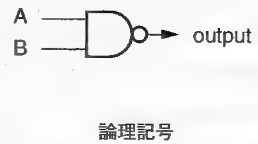
この規則で「近い」が何を意味するか定義できたとすれば，V_{dd}に近い値を「真」，V_{SS}に近い値を「偽」と関連づけることができます．すると回路は，NANDブール論理関数によって表現できます．

$$output = \neg(A \wedge B) \quad \text{（式1）}$$

この数式を4個のトランジスタで高信頼性の回路として実現しようとすると，工学的にはかなりたいへんな設計となります．しかし，論理設計者は，論理ゲートを使った論理的な設計をするだけで，十分に高い信頼性



〔図1.2〕2入力CMOS スタティックNAND
ゲートのトランジスタ回路



A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

真理値表

〔図1.3〕論理記号とNANDゲートの真理値表

のシステムを実現できるのです．論理設計者が扱う概念を図1.3に示しますが，それは論理ゲートを次のような「視点」から見たものです．

(1) 論理記号

この図で示されるのは回路図上でNANDゲート機能を表す記号です．ほかの論理ゲートもよく似た記号を使います(たとえば，出力の丸印を取ると，逆の出力機能をもつANDゲートになる．その他の例は，本書巻末の「付録A：コンピュータ論理」を参照)．

(2) 真理値表

ゲートの論理機能を表し，いろいろな目的に使うゲートについて論理設計者が知らなくてはならないことの全部を示しています．4組のトランジスタよりずっと単純になります(ブール変数での慣例に従い，この真理表で真は“1”，偽は“0”で表す)．

◆ゲートの抽象化

「ゲート」という抽象化の導入により，多数のトランジスタを使った回路設計の工程が単純化されるだけなく，論理回路設計者は，もはやゲートがトランジスタから構成されることを考える必要もなくなります．FET(CMOSプロセスで利用できるトランジスタ)，バイポーラ・トランジスタ，電気式リレー，流体論理など，ゲートがどれで実現されていようとも，論理回路は同じ論理動作をします．

実装技術の善し悪しは，回路の性能には影響しますが，機能には影響しません．論理回路の設計者がトランジスタの等式を理解する必要がないようにするためには，トランジスタ・レベルの回路設計者は，ゲートの抽象化を完璧なまでにサポートする義務があります．

◆抽象化のレベル

長年，論理ゲートに親しんできた読者にとっては，話がまどろっこしいかもしれません．しかし，ゲート・レベルの抽象化で説明した基本原則は，コンピュータ工学の各レベルで繰り返し使われており，本節で考えている複雑さに対処するためには絶対に必要です．

あるレベルの部品を集めて，それらの全体としての主要な振る舞いを抽出し，その上位のレベルで不必要な詳細な動作情報を隠してしまえば，複雑な要素の記述をほんの数ステップで記述することができます．たとえば，ここで扱うゲート・モデルのように，各レベルが一つ下レベルの4コンポーネントから構成されるとすると，1個のトランジスタから100万個のトランジスタからなるマイクロプロセッサまで，たった10ステップで達します(なぜなら，$4^{10}=1,048,576$)．多くの場合，4以上のコンポーネントを扱いますから，ステップはかなり削減されます．

ハードウェア・レベルでの一般的な抽象化の階層は，以下のようになります．

第1層 トランジスタ

第2層 論理ゲート，メモリ・セル，特別な回路

第3層 一ビット加算器，マルチプレクサ，デコーダ，フリップフロップ

第4層 ワード加算器，マルチプレクサ，デコーダ，レジスタ，バス

第5層 ALU（算術論理演算装置），バレル・シフタ，レジスタ・バンク，メモリ・ブロック

第6層 プロセッサ，キャッシュおよびメモリ管理構成

第7層 プロセッサ，ペリフェラル・セル，キャッシュ・メモリ，メモリ管理ユニット

第8層 集積システム・チップ

第9層 プリント回路基板

第10層 携帯電話，パソコン，エンジン制御装置

設計がハードウェアで表現されている場合，抽象化レベルの観点から設計を理解する過程は，きわめて具体的です．しかし，この方法はハードウェアだけではなく，ソフトウェアを理解する場合にも重要となります．ソフトウェア設計の抽象化については，後で述べます．

◆ゲート・レベルの設計

論理ゲートから一歩進んで，複数のゲートから構成される基本的な機能ライブラリを組み立てましょう．典型的な機能としては，上述したように1ビット幅の加算器，マルチプレクサ，デコーダ，フリップフロップがあります．本書の主な内容はプロセッサ・コアの使用に関するもので，本書を活用されようとする読者なら，従来の論理設計の知識はもっていると思います．論理設計になじみの少ない読者や過去の知識を思い出す必要のある読者のために，「付録A：コンピュータ論理」に必要な事項を簡潔に説明しています．

- ブール代数と記号
- 2進法（バイナリ）数値
- 2進法の加算
- マルチプレクサ
- クロック
- シーケンス回路
- ラッチとフリップフロップ
- レジスタ

もし上記の用語で知らないものがあれば，この付録を一読して意味を把握してみてください．

付録Aでは，単純な論理ゲートを使って回路の機能を説明しましたが，たいていの場合はほかのトランジスタ回路を使うより効率的なCMOS回路で実装しています．CMOSチップ上で実装できる相補型トランジスタを使えば，論理設計にとって必要なことはすべてできます．また，新しいトランジスタ回路は次々に発表されています．

詳しくは，論理設計に関する記述を参照してください．参考書は，付録Cの「参考文献」に列記してあります．

1.3 MU 0 ── 単純プロセッサ

単純なプロセッサは，以下の基本コンポーネントから構成されます．

(1) **プログラム・カウンタ(PC)**：進行中の命令のアドレスを保持するレジスタ．
(2) **アキュムレータ(ACC)**：処理中の演算の結果を保持する単一レジスタ．
(3) **算術論理演算装置(ALU)**：加減算，インクリメントなど各種バイナリ演算を実行するユニット．
(4) **命令レジスタ(IR)**：命令が実行されている間，その命令を保持するレジスタ．
(5) 望む結果を得るため，上記ユニットを利用する命令デコーダと制御論理回路．

限定された部品からできているので，実装する命令セットも制限されています．この方法は，プロセッサ設計の原理を説明するためにマンチェスター大学で昔から用いられてきました．当時のマンチェスター大学で設計されたプロセッサは，MUn（nは数字で1〜6があった）と命名されていたので，この単純マシンはMU0と呼ばれました．これは講義用に開発したもので，研究用に大学で開発した大規模マシンの一つではありませんでしたが，初代マンチェスター・マシンに似ていて学生によって，さまざまな形で実装されました．

◆MU0命令セット

MU0は，12ビットのアドレス空間をもつ16ビットのマシンで，最大メモリ8Kバイトまでで個別にアドレス可能な4,096個の（16ビット幅）メモリ位置をもちます．命令は16ビット長で，図1.4に示すように4ビットのオペレーション・コード（以下，OPコード）と12ビットのアドレス・フィールド（S）をもちます．もっとも単純な命令セットは，16（=4ビット）あるOPコードのうち8個だけを使いました（表1.1）．

たとえば，

```
ACC := ACC + mem16[S]
```

のような命令は，「アドレスがSのメモリ位置の内容（16ビット幅）をアキュムレータに加える」という意味です．命令は，メモリ・アドレス0からPCを変更する命令が実行されるまで，連続するメモリ・アドレスからフェッチされ（取り込まれ），実行します．新しいアドレスのフェッチは，ジャンプ（JMP）命令で与えられます．

4ビット	12ビット
OPコード	S

〔図1.4〕MU0命令形式

〔表1.1〕MU0命令セット

命令	OPコード	実行
LDA S	0000	$ACC := mem_{16}[S]$
STO S	0001	$mem_{16}[S] := ACC$
ADD S	0010	$ACC := ACC + mem_{16}[S]$
SUB S	0011	$ACC := ACC - mem_{16}[S]$
JMP S	0100	$PC := S$
JGE S	0101	if $ACC \geq 0$ $PC := S$
JNE S	0110	if $ACC \neq 0$ $PC := S$
STP	0111	stop

◆MU0論理設計

この命令セットをどう実装するかを理解するために，論理シーケンスに基づいて設計プロセスを見てみましょう．ここでは，設計を二つのコンポーネントに分けます．

(1) **データパス部**

アキュムレータ，プログラム・カウンタ，ALU，命令レジスタなど，多数のビットを並列的に転送，保存，処理しているすべてのコンポーネントはデータパスの一部とします．設計には，レジスタ，マルチプレクサなどに基づいた，レジスタ転送レベル（RTL）設計を用います．

(2) 制御論理部

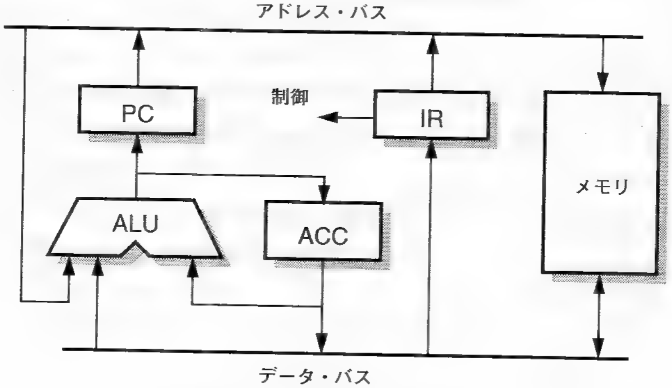
データパスに当てはまらないものは制御論理の一部分と見なされ，有限状態マシン（FSM：Finite State-machine　以下，ステートマシン）の方法を使って設計します．

◆データパス設計

MU0命令セット実装のために基本コンポーネントを連結するには，多くの方法があります．多くの選択肢の中から正しいものを選択をするには，指針となる原則が必要です．ここでは，メモリが制限要素になること，メモリ・アクセスにはつねに1クロック・サイクルがかかるという原則に従います．実際の実装に向けて考えていきましょう．

- 各命令は，必要なメモリ・アクセス回数と同数のクロック・サイクルを使う

表1.1を振り返ってみましょう．最初の4命令は，それぞれ2回のメモリ・アクセスが必要です（一つは命令自体をフェッチするため，一つはオペランドをフェッチもしくは保存するため）．また，後半の4命令は，オペランドが不要なので，1サイクルで実行できます（実際には，プロセッサを永久に止めてしまうSTP命令について深く考える必要はない）．したがって，これらの命令を2または1クロック・サイクルで完結するのに十分なリソースをもつデータパス設計を考える必要があります．適切なデータパスは図1.5のようになります．（このデータパスで，PC専用のインクリメンタを必要と思った読者は，PCを変更しない命令が，すべて2サイクルかかることに注目してほしい．このうちの1サイクルで，このALUがPCをインクリメントする．）



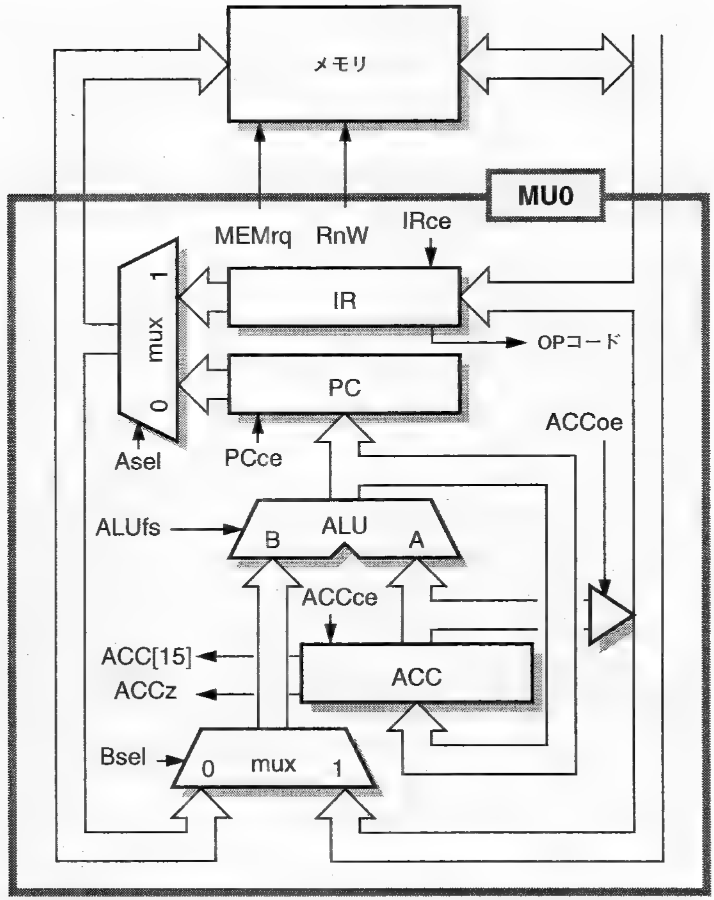
〔図1.5〕MU0データパスの例

◆データパス処理

ここで開発する設計では，命令は命令レジスタに着いたときに実行が開始されます．命令レジスタに着くまでは，何の命令かわかりません．したがって，命令は次の2段階で実行しますが，最初の段階は，省略してもよいでしょう．

(1) メモリにあるオペランドにアクセスし，対応する処理を実行する

まず命令レジスタのアドレスが発行されたあと，オペランドがメモリから読み出され，ALUの中のアキュ



〔図1.6〕MU0レジスタ転送レベル構成

ムレータと組み合わされた後アキュムレータに再度書き込まれるか，もしくはアキュムレータの内容がメモリに格納されます．

(2) 次に実行する命令をフェッチする

次の命令をフェッチするため，PCまたは命令レジスタ内のアドレスが発行されます．どちらの場合もアドレスはALUでインクリメントされ，インクリメントされた値はPCに保存されます．

◆初期化

プロセッサは，レジスタなど内部が既知状態で開始されなければなりません．通常，*reset*信号が入力され，既定のアドレスから命令が実行されることが必要です．MU0は，アドレス000_{16}から実行するように設計します．方法はいくつかありますが，*reset*信号を使ってALU出力をゼロにし，これをPCレジスタに入力するのもその一つです．

◆レジスタ転送レベル(RTL)設計

次に，データパスがすべての処理を実行するのに必要な制御信号を正確に決定します．レジスタはすべて，

表1.2 MU0制御論理

	入力					出力									
命令	OPコード	Reset	Ex/ft	ACCz	ACC15	Asel	Bsel	ACCce	PCce	IRce	ACCoe	ALUfs	MEMrq	RnW	Ex/ft
Reset	xxxx	1	x	x	x	0	0	1	1	1	0	= 0	1	1	0
LDA S	0000	0	0	x	x	1	1	1	0	0	0	= B	1	1	1
	0000	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
STO S	0001	0	0	x	x	1	x	0	0	0	1	x	1	0	1
	0001	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
ADD S	0010	0	0	x	x	1	1	1	0	0	0	A+B	1	1	1
	0010	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
SUB S	0011	0	0	x	x	1	1	1	0	0	0	A-B	1	1	1
	0011	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
JMP S	0100	0	x	x	x	1	0	0	1	1	0	B+1	1	1	0
JGE S	0101	0	x	x	0	1	0	0	1	1	0	B+1	1	1	0
	0101	0	x	x	1	0	0	0	1	1	0	B+1	1	1	0
JNE S	0110	0	x	0	x	1	0	0	1	1	0	B+1	1	1	0
	0110	0	x	1	x	0	0	0	1	1	0	B+1	1	1	0
STOP	0111	0	x	x	x	1	x	0	0	0	0	x	0	1	0

入力クロックの立ち下がりエッジで状態が変化し，必要なら特定のクロック・エッジで変化しないようにするための制御信号をもつものとします．たとえばPCは，*PCce*が‘1’ならクロック・サイクルの終わりで変化しますが，‘0’なら変化しません．

これにふさわしいレジスタ構成は，図1.6のとおりです．ここに，全レジスタのイネーブル，ALUへの機能選択ライン（正確な数字と解釈は後で決定する），二つのマルチプレクサ用の選択制御ライン，ACC値をメモリに送る3ステート・ドライバの制御，メモリ要求（*MEMrp*），読み書き（*RnW*）制御ラインが示してあります．OPコード・ビットや，ACCがゼロか負かによって条件ジャンプ命令を制御する信号までを表示し，データパスから制御論理への出力も示してあります．

◆制御論理

制御論理は，データパスからの必要な制御入力を使って，単純に現在の命令をデコードし，適切なデータパス制御信号を生成するだけです．制御論理はステートマシンであり，本来なら状態遷移図から設計を始めるべきですが，この場合ステートマシンが非常に小さいので図を書く必要はありません．実装に必要なのは「フェッチ」と「実行」のたった二つの状態だけです．つまり，1ビット（Ex/ft）で足ります．

制御論理は，表1.2のように表形式で示すことができます．この表では，‘x’は「don't care」状態を示します．ALU機能選択コードが割り当てられると，表はPLA（programmable logic array）としてそのまま実装できます．もしくは，組み合わせ論理に変換し，標準ゲートを使って実装できます．

表1.2を見ると，単純な規則に気づきます．プログラム・カウンタと命令レジスタのクロック・イネーブル（*PCce*と*IRce*）はつねに同じです．つまり，新しい命令がフェッチされるたびにALUは次のプログラム・カウ

ンタの値を計算しており，これがラッチされるべきです．これは当然のことで，これらの制御信号は一つに統合できます．同様に，アキュムレータがデータ・バスを駆動している（*ACCoe*が‘H’）とき，つねにメモリは書き込み（*RnW*が‘L’）をしますから，もう一方の信号にインバータを用いれば生成できることになります．

これらの単純化が終わると，制御論理設計はほぼ終了です．あとは，ALU機能のエンコードを決定するだけです．

◆ALU設計

図1.6に示したレジスタ転送レベル機能のほとんどは，そのまま単純な論理として実装しています（「付録A：コンピュータ論理」を参照）．ただし，MU0 ALUは，付録に書いてある単純な加算器よりは少しだけ複雑になります．

要求されるALU機能は，表1.2に示すとおりです．機能は，

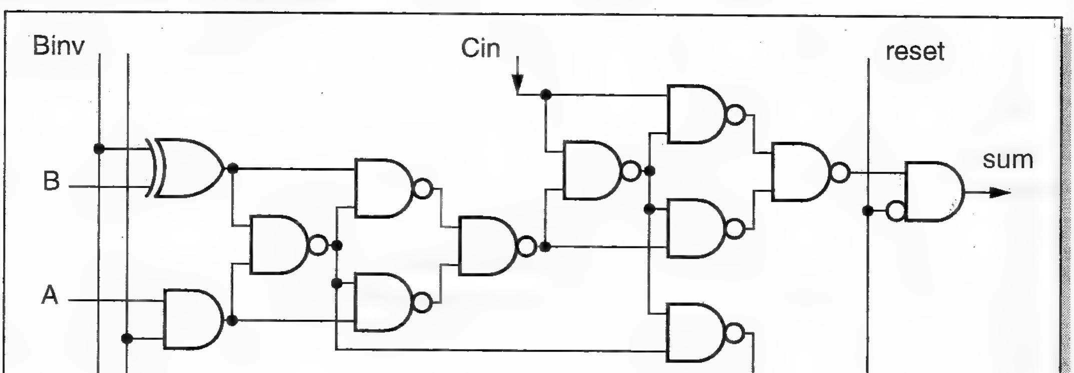
$A+B$，$A-B$，B，$B+1$，0

の五つです．最後の0は，リセットがアクティブなときにのみ使用します．したがって，リセット信号はこの機能を直接制御し，制御論理は残りの四つを選択する2ビット機能選択コードのみを生成すればよいのです．

主ALU入力がAおよびBオペランドであれば，全機能は従来のバイナリ加算器を増やすだけで作ることが可能です．

- $A+B$は，通常の加算器出力（キャリインは0とする）
- $A-B$は，Bを反転，キャリインを1に強制することで$A+\overline{B}+1$として実現
- Bは，A入力とキャリインを0に強制することで実現
- $B+1$は，A入力を0，キャリインを1に強制することで実現

ALUのゲート・レベル論理は，図1.7のとおりです．*Aen*はAオペランドをイネーブル，または0に強制します．*Binv*はBオペランドを反転させるかどうかを制御します．各ビットからのキャリアウト（*Cout*）は次のキャリイン（*Cin*）に接続します．最初のビットへのキャリインはALU機能選択（*Aen*と*Binv*）によって制御され，最後のビットからのキャリアウトは使用されません．マルチプレクサ，レジスタ，制御論理，バス・バッファ（アキュムレータ値をデータ・バスに乗せる）と合わせるとプロセッサは完成です．標準メモリを加えれば，実際に使えるコンピュータとなります．



〔図1.7〕1ビットについてのMU0 ALUゲート・レベル回路

◆MU0の拡張

MU0は，非常に単純なプロセッサなので，高級言語コンパイラのサポートはできませんが，プロセッサ設計の基本原則を説明するには最適です．最初のARMプロセッサを開発するのに用いた設計プロセスも，複雑さに違いはありますが，原則は同じです．マイクロコード制御論理に基づくMU0設計も開発されており，インデクス修飾アドレシングを追加拡張しています．普通のプロセッサ設計と同じように，MU0は，命令空間に余白を残してあり，命令セットの将来の拡張に備えています．

MU0を実用的なプロセッサに変えるには，まだまだ手を加えなければなりません．重要な拡張は以下のとおりです．

- アドレス空間の拡張
- アドレシング・モードの追加
- サブルーチン機構をサポートするためのPC保存
- レジスタ追加，割り込みに対応，など．

ただし，コンパイラ支援を考えた高性能プロセッサを設計するのが目的なら，このようなスタートは良いとはいえません．

1.4　命令セット設計

MU0命令セットが高性能プロセッサに適していないとすれば，何が適しているのでしょうか？　最初の原理に戻って，二つの数を足して和を求める命令などの基本的な演算処理を見てみましょう．

◆4アドレス命令

きわめて一般的な形のこの命令に必要なのは，

(1) 他の命令と区別するためのOPコードを指定するfビット
(2) オペランド・アドレス（ソース）1を指定するnビット
(3) オペランド・アドレス（ソース）2を指定するnビット
(4) 結果をおく位置（デスティネーション；行き先）を指定するnビット
(5) 次に実行する命令のアドレスを指定するnビット

の五つです．そのような命令のアセンブリ言語書式は以下のようになります．

```
ADD  d, s1, s2, next_i ; d := s1 + s2
```

このような命令は，図1.8に示すようにバイナリ形式でメモリ上に表せます．この書式は命令ごとに$4n+f$ビットが必要です．つまり各オペランドがnビット，ADDを指定するOPコードがfビットです．

fビット	nビット	nビット	nビット	nビット
function	op 1 addr.	op 2 addr.	dest. addr.	next_i addr.

〔図1.8〕4アドレス命令フォーマット

◆3アドレス命令

各命令に必要なビット数を減らす第一の方法は，次に実行する命令を暗黙に指定（インプリシット）するこ

とです（分岐命令は，命令シーケンスを明示的に変更する機能をもつため除く）．命令サイズをPCに加えることによってデフォルトの次の命令を知ることができれば，以下のようなアセンブリ言語書式をもった3アドレス命令が得られます．

```
ADD  d, s1, s2,    ; d := s1 + s2
```

このような命令のバイナリ表現は図1.9のとおりです．

fビット	nビット	nビット	nビット
function	op 1 addr.	op 2 addr.	dest. addr.

〔図1.9〕3アドレス命令フォーマット

◆2アドレス命令

命令のストアに必要なビットをさらに節約するには，デスティネーション・レジスタをソース・レジスタの一つと同じにします．アセンブリ言語書式は以下のとおりです．

```
ADD  d, s1,   ; d := d + s1
```

バイナリ表現は，図1.10のように減ります．

fビット	nビット	nビット
function	op 1 addr.	dest. addr.

〔図1.10〕2アドレス命令フォーマット

◆1アドレス命令

デスティネーション・レジスタが暗黙に指定される場合，それはアキュムレータと呼ばれます（たとえば，1.3節「MU0」を参照）．命令は一つのオペランドを指定するだけですみます．

```
ADD  s1       ; accumulator := accumulator + s1
```

バイナリ表現は，図1.11のようにさらに単純化されます．

fビット	nビット
function	op 1 addr.

〔図1.11〕1アドレス（アキュムレータ）命令フォーマット

◆0アドレス命令

最後に，評価スタックを使って，アーキテクチャはすべてのオペランド参照を暗黙に指定することが可能です．アセンブリ言語書式は以下のとおりです．

```
ADD   ; top_of _stack := top_of_stack + next_on_stack
```

バイナリ表現は，図1.12のとおりです．



〔図1.12〕0アドレス命令フォーマット

◆nアドレス使用例

これらの命令形式は全部，プロセッサ命令セット内で使われています．ただし，4アドレス形式は，一部のマイクロコード設計で内部的に使われているものの，機械語レベルの命令セットにしては必要なほど不経済となるために使用されていません．以下に例をあげます．

- Inmos社トランスピュータは，0アドレス評価スタック・アーキテクチャを使用
- 前述のMU0は，単純な1アドレス・アーキテクチャを使用
- ARMプロセッサで高密度コード向けに用いられているThumb命令セットは，ほとんど2アドレス形式（第7章参照）．
- 標準のARM命令セットは，3アドレス・アーキテクチャを使用

◆アドレス

MU0アーキテクチャのアドレスは，メモリ位置を直接オペランドに記述する絶対アドレスを用いています．しかし，ARMの3アドレス命令形式の3アドレスとは，レジスタ指定子でありメモリ・アドレスではありません．通常，3アドレス・アーキテクチャという用語は，2ソース・オペランドとデスティネーションを別個に指定する命令セットを指します．ただし，たいていの場合，取り得る値には制限があります．

◆命令タイプ

ここまで，ADD命令を指定する多様な方法を見てきました．完全な命令セットは，メモリ内でオペランドにしたがって数値演算をするだけでは足りません．一般命令セットは，以下のカテゴリの命令にも対応する必要があります．

- 加算，減算，乗算などのデータ処理命令
- データをメモリのある場所から別の場所へ，またはメモリからプロセッサのレジスタへコピーするデータ移動命令
- データ値に応じて，プログラムのある部分から別の部分へ実行をスイッチするフロー制御命令
- OS機能を実行するために特権モードに切り替えるなど，プロセッサの実行状態を制御する特別命令

上記のカテゴリの複数に当てはまる命令もあります．たとえば「非0ならデクリメントかつ分岐」命令はプログラム・ループを制御するのに有効ですが，ときにはループ変数についてデータ処理も行い，またフロー制御機能も果たします．同様に，メモリ内のアドレスからオペランドをフェッチし，結果をレジスタに置くデータ処理命令は，データ転送機能を果たしているといえます．

◆直交型命令

命令セットの構築上，各選択がほかの選択から独立している場合，その命令セットは直交型（orthogonal）であると言います．加算と減算は類似の演算ですから，同様の状況で使用することがあります．加算がレジスタ・アドレスに3アドレス形式を用いるのであれば，減算も同様で，どちらの場合も使用するレジスタに特別な制約はないはずです．

直交型命令セットは，アセンブリ言語のプログラマにとって学習しやすく，コンパイラ設計者にとってター

ゲットにするのが簡単でもあります．ハードウェア実装も通常は効率的です．

◆アドレシング・モード

データ処理または転送命令のためにオペランドにアクセスする際，必要な位置を指定するには標準的な手法がいくつかあります．多くのプロセッサは，下記の**アドレシング・モード**のうちいくつかをサポートしています（全部をサポートしているものは，ほとんどない）．

(1) **イミディエート・アドレシング**： 求める値が命令のバイナリ値として表現される．
(2) **絶対アドレシング**： 命令が求める値の完全バイナリ・アドレスを指定する．
(3) **間接アドレシング**：求める値のバイナリ・アドレスを格納するメモリ位置のバイナリ・アドレスを，命令が指定する．
(4) **レジスタ・アドレシング**： 求める値はレジスタ内にあり，命令はレジスタ番号を指定する．
(5) **レジスタ間接アドレシング**： メモリ内の値のアドレスを格納するレジスタ番号を，命令が指定する．
(6) **ベース・オフセット・アドレシング**： 命令が，レジスタ（ベース：基底）と，ベースに加えてメモリ・アドレスを作るバイナリ・オフセットを指定する．
(7) **ベース・インデクス・アドレシング**： 命令が，ベース・レジスタと，ベースに加えてメモリ・アドレスを作るもう一つのレジスタ（**インデクス**）を指定する．
(8) **ベース・拡張インデクス・アドレシング**： 上記と同じ．ただしインデクスはベースに加える前に定数（通常はデータ項目のサイズで，2の累乗であることが多い）を掛ける．
(9) **スタック・アドレシング**： 暗黙に指定される，すなわち指定されたレジスタ（**スタック・ポインタ**）がメモリのある場所（**スタック**）を指していて，データはLIFO（後入れ先出し）方式で書き込んだり（**プッシュ**），読み出したり（**ポップ**）します．

モードの呼び名については，各プロセッサ・メーカによって上記と異なることがあるので注意してください．多段の間接化や，より多くのベースとインデクスとオフセットを加えるなどして，リストは半永久的に拡張できます．しかし，一般的なアドレシング・モードの多くは上記のリストにあります．

◆フロー制御命令

プログラムがデフォルトの（通常の）命令シーケンスから外れなければならないとき，フロー制御命令がプログラム・カウンタ（PC）を明示的に変更します．そのうちもっとも単純なものは，「分岐」または「ジャンプ」と呼ばれます．多くの分岐は，比較的短い範囲しか必要としないため，一般的な形式は「PC相対」分岐です．典型的なアセンブリ言語書式は以下のとおりです．

```
        B       LABEL
        ・・・
LABEL   ・・・
```

ここで，分岐が実行されたときにPCの値に加えなければならない変位をアセンブラが計算し，PCがLABELを指すようにします．分岐の最大範囲は，バイナリ形式で表した変位のビット数によって決まります．分岐が範囲を出ている場合には，アセンブラがエラーを報告します．

◆条件分岐

ディジタル信号処理（DSP）プログラムは，決まった命令シーケンスを永久に実行していればよいのですが，

汎用プロセッサは普通，データ値に応じてプログラムを変更することが求められます．一部のプロセッサ（MU0など）では，汎用レジスタの値によって，以下のような命令を通して分岐をするかどうかが制御されます．

- 特定のレジスタが0のとき分岐（非ゼロのとき，負のとき，など）
- 二つの特定のレジスタが等しいとき分岐（もしくは，等しくないとき）

◆条件コード・レジスタ

しかし，もっともひんぱんに使われるメカニズムは，条件コード・レジスタに基づいています．条件コード・レジスタは，プロセッサ内では特別な目的をもったレジスタです．データ処理命令が実行されると（もしくは，特別な命令の場合，すなわち条件コード・レジスタをイネーブルする命令の場合），条件コード・レジスタは結果が0だったのか，あふれたのか，桁上げ出力したのか，などを記録します．そして，条件分岐命令はその実行の際，条件コード・レジスタの状態によって制御されます．

◆サブルーチン・コール

サブプログラムをコールするために分岐が実行されることがあります．これは，命令シーケンスが，サブプログラム終了時，コールしているシーケンスに**復帰**しなければならない場合です．サブプログラムは，さまざまな場所からコールされる可能性があるので，コールしているアドレスの記録は残しておかなければなりません．これには，以下のように多くの方法があります．

1) コールしているルーチンが適切な復帰アドレスを分岐を実行する前に割り出して標準メモリ位置に置いておき，サブプログラムが復帰アドレスとして使用できるようにする．
2) 復帰アドレスをスタックにプッシュする．
3) 復帰アドレスをレジスタに置く．

サブプログラム・コールはかなり一般的なので，効率的に行うために多くのアーキテクチャは特定の命令をもっています．

サブプログラム・コールは単純な分岐より遠く離れたメモリにジャンプしなければならない場合が多いので，別個に扱うのが妥当です．条件的でないことも多くあります．必要なら，無条件コールとそれをバイパスするように逆の条件での分岐命令を挿入しておくことで条件付きのサブプログラム呼び出しがプログラムできます．

◆サブプログラム復帰

復帰命令は，戻りアドレスを保存していた場所から（メモリ，場合によってはスタック，レジスタ）PCに移動します．

◆システム・コール

システム・コールもフロー制御命令のカテゴリの一つです．これは，OSルーチンへの分岐で，実行するプログラムの**特権レベル**の変更をともないます．プロセッサの一部の機能，たとえば周辺との入出力は，ユーザ・コードによるアクセスから守られています．したがって，これらの機能にアクセスしようとするユーザ・プログラムは，システム・コールを使う必要があります．

システム・コールを使えば，制御された方法で保護バリアをくぐり抜けることができます．よく設計されたプロセッサなら，あるユーザのプログラムを他ユーザ（悪意をもっている場合もある）の攻撃から守れるマルチユーザ・システムを書くことができます．そのためには，悪意をもったユーザがシステム・コードを変更することができないこと，そして保護されている機能にアクセスしようとした場合には，要求する機能が承認さ

れたものであるかをシステム・コードによって確認されるようになっていなければなりません．

これは，ハードウェアとソフトウェアの設計にかかわる複雑な領域です．多くの組み込み型システム（多くのデスクトップ・システムも）は，ハードウェアの保護機能をフルに使っていませんが，保護システム・モードをサポートしないプロセッサは，保護機能を要求するアプリケーションを受け入れることができません．このような経緯から，いまではほとんどのマイクロプロセッサがこの機能を備えています．命令セットの基本設計を理解するうえで，安全なOSをサポートする意味を全部理解する必要はありません．しかし保護機能を実現する能力をもつことの意味をよくわかっていないと，商用プロセッサのアーキテクチャの特徴のいくつかは意味を理解できないので，読者がこれらに関して詳しくないとしても問題点については知っておくべきでしょう．

◆例外

フロー制御命令の最後のカテゴリに入るのは，プログラマの当初の意図ではなく，プログラムが併発した予定外の（望ましくないことも多い）現象の結果が引き起こした制御フローの変化です．メモリ・サブシステムが失敗を検知すると，メモリ位置にアクセスできない可能性があります．そしてこの問題から回復するためには，プログラムは予定された処理フローを外れることが必要です．

予定外の制御フローの変化を**例外**（exception）と呼びます．

1.5 プロセッサ設計におけるトレードオフ

プロセッサ設計でもっとも神経を使うのは，プログラマに有用な機能をサポートする命令セットを定義すると同時に，できる限りその実現を効率的にすることです．そして，同じ命令セットが将来のより洗練した実装にも効率的に対応できれば，なお望ましいことです．

一般的にプログラマは，問題に対して適切な概念を扱える方法をサポートしている高級言語を使って，プログラムをできるだけ抽象的に表現しようとします．現代の機能分割やオブジェクト指向言語への流れは，すでに古くなっているとはいえ典型的な機械語命令よりはずっと抽象度の高いC言語などより，さらに高い抽象度へと移行しています．

高級言語構造と機械語命令の間の**セマンティック・ギャップ**（意味論的不一致）を埋めるのは**コンパイラ**です．コンパイラは，高級言語のプログラムを機械語命令のシーケンスに翻訳するコンピュータ・プログラムです．プロセッサ設計者は，コンパイラを作成しやすいような命令セットを定義し，プログラマが手で直接問題を解決しなくても済むように心がけなければなりません．それでは，どのような命令セットが良いコンパイラ・ターゲットとなるのでしょうか？

◆複雑命令セット・コンピュータ

1980年以前は，コンパイラが埋めなければならないセマンティック・ギャップを小さくしようと，命令セットの設計が複雑化する傾向にありました．一つの命令であるプロシージャのエントリやエグジット命令が，何サイクルにもわたって一連の複雑な動作が実行される命令から構成されていました．プロセッサは，その洗練度とアドレシング・モード数，データ型数などを重視して開発されていました．

この傾向は，1970年代に開発されたマイクロプロセッサから見ることができます．このコンピュータは，比較的遅いメイン・メモリを多くの単純なICを使ってプロセッサに接続していました．プロセッサは，メイン・メモリより高速のマイクロコードROMで制御されていたため，ひんぱんに使用する処理はマイクロコー

ド・シーケンスとして実装し，メイン・メモリからいちいち命令をフェッチしないほうが妥当でした．

1970年代，マイクロプロセッサの性能は向上を続けましたが，このときはいかにして単一のチップに多くのトランジスタを載せるかに重点をおいた最先端の半導体技術に頼っていました．その発展は，コンピュータ業界というより半導体業界におけるものだったといえます．結果として，とくに実装に関する技術の需要にもかかわらず，マイクロプロセッサ設計はアーキテクチャ・レベルでの新しい思想を欠いていました．設計者は，せいぜい実装技術がまったく異なるミニコンピュータ業界からアイデアを得るだけでした．とりわけ，複雑なルーチンに必要なマイクロコードROMがチップ上で場所をとっていたため，性能向上にあてる場所がほとんどありませんでした．

この傾向で生み出されたのが，1970年代の複雑な命令セット・コンピュータ（**CISC**；Complex Instruction Set Computer）であり，それはミニコンピュータの命令セットをもちながら，シリコン・リソースの非常な制約を強いられたマイクロコンピュータでした．

◆RISC革命

命令セットがますます複雑化する中で，縮小命令セット・コンピュータ（**RISC**；Reduced Instruction Set Computer）は誕生しました．RISCの概念は，ARMプロセッサの設計に大きく影響しましたので，RISCはARMの別名といってもいいぐらいです．でも，RISCやARMの詳細を考える前に，プロセッサが何をし，また高速で動作するためにどのように設計されるのかについて基本的な事項をざっと見てみましょう．プロセッサの命令セットと高級言語との間のセマンティック・ギャップを埋めることで効率的なコンピュータができないとすれば，設計者はほかにどんな方法をとればよいのでしょうか？

◆プロセッサの仕事

高速のプロセッサを作るには，プロセッサが何に時間を費やしているのかを理解する必要があります．コンピュータが計算，つまりユーザ・データの算術処理に時間を費やしているというのは誤解です．この意味での「計算」時間は，じつは非常に少ないのです．たしかに計算はかなりありますが，多くはアドレスに関して関連データ項目やプログラム・ルーチンの位置を求めるものです．そしてユーザのデータを見つければ，それを処理して変化させるというよりは，移動させるのがおもな仕事となります．

命令セット・レベルで，命令の使用頻度を測ることは可能です．実行する命令の頻度，つまり**動的**（ダイナミック）な測定は，バイナリ形式の命令タイプを数えるだけの**静的**（スタティック）頻度測定より重要です．典型的な統計を表1.3に示します．これらの数値は，ARM命令エミュレータでプリント・プレビュー・プログラ

〔表1.3〕命令の動的な使用頻度

命令タイプ	動的使用率
データ移動	43%
フロー制御	23%
算術演算	15%
比較	13%
論理演算	5%
その他	1%

ムを動作して出したものですが，他のプログラムや命令セットにも広く当てはまります．

このサンプル統計でわかることは，プロセッサ・レジスタとメモリ間，またはレジスタ同士の間のデータ移動に関する命令を最適化することがもっとも重要だということです．移動に関する命令は，実行する命令全体の半分を占めているのです．

2番目に頻度が高いのは，分岐やプロシージャ・コールなどのフロー制御命令で，4分の1を占めます．算術的処理と比較はそれぞれ15%程度です．

プロセッサが何に時間を費やしているかを見ましたから，あとは高速化を考えましょう．もっとも重要な技術はパイプラインです．そして，キャッシュ・メモリの使用です(10.3節も参照)．3番目はスーパスカラ命令実行ですが，これは非常に複雑でARMのプロセッサには使われていないので，本書では説明しません．

◆パイプライン

プロセッサは個別の命令を一連のステップに分けて実行します．典型的なシーケンスは次のとおりです(カッコ内のステージ名は図中で用いる)．

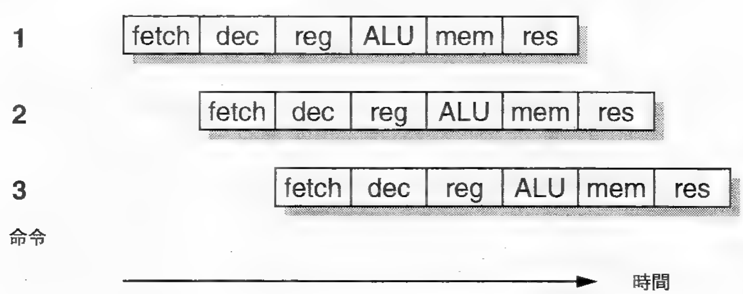
(ステップ1) メモリから命令をフェッチ(fetch)
(ステップ2) 命令の内容を知るためにデコード(dec)
(ステップ3) レジスタ・バンクから必要とされるオペランドをアクセス(reg)
(ステップ4) オペランドを組み合わせて結果もしくはメモリ・アドレスを形成(ALU)
(ステップ5) 必要に応じてデータ・オペランドを取りにメモリにアクセス(mem)
(ステップ6) 結果をレジスタ・バンクに再度書き込み(res)

すべての命令に上記の全ステップが必要とはかぎりませんが，ほとんどの命令には上記のほとんどが必要だといえます．これらの各ステップは，それぞれ異なったハードウェア部分を使う傾向があります．たとえば，ALUはおそらくステップ4でしか使われません．したがって，前の命令が終わるまで次の命令を開始しないのであれば，どのステップでもプロセッサ・ハードウェアのほんの一部分しか使わないことになります．

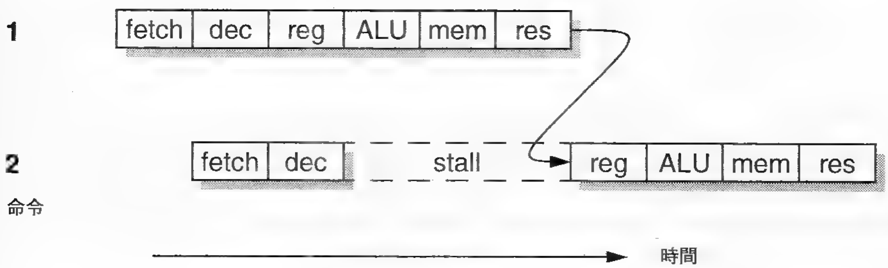
ハードウェア資源を効率よく使い，プロセッサ・スループットを向上させるには，現在の命令が終わる前に次の命令を開始すればよいことは明らかです．これをパイプライン(pipeline)といい，汎用プロセッサで並列性を生かす非常に効果的な方法です．

上記の処理シーケンスで，一つの命令がステップ1を終わりステップ2に移動すると同時に，次の命令がステップ1を開始するようにプロセッサは設定されています．これを図1.13に示します．

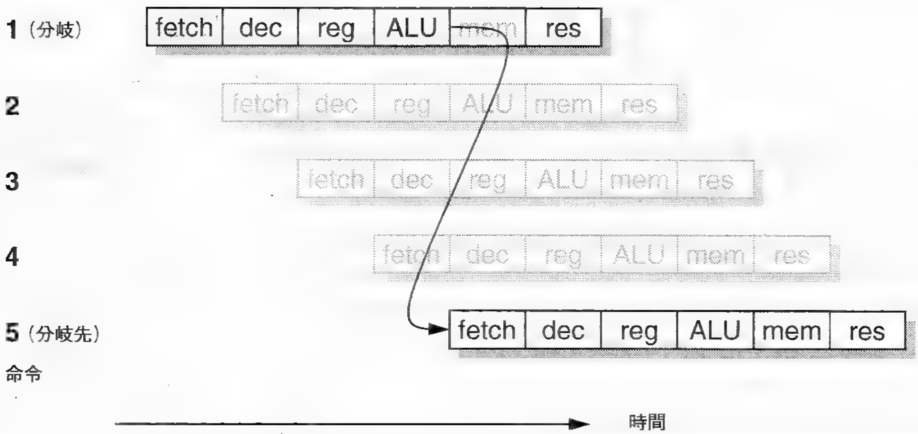
理論上，このようなパイプラインは一つずつ命令を実行するのに比べて6倍も高速化します．ただし実際は，次のような理由でそれほどうまくはいきません．



〔図1.13〕パイプライン方式の命令実行



〔図1.14〕リード・アフタ・ライト・パイプライン障害



〔図1.15〕パイプライン方式での分岐の動き

◆パイプライン障害

コンピュータ・プログラムにおいて，ある命令の結果が次の命令のオペランドとして使われることがよくあります．この場合，命令1の結果は命令2がオペランドを読み込んだ時点でまだ確定していませんから，図1.13に示したパイプライン処理が崩れてしまいます．命令2は命令1の結果が確定するまで停止しなければならず，図1.14に示したような動きをします．これがリード・アフタ・ライト（read-after-write）パイプライン障害です．

分岐命令は，もっと悪いパイプライン障害となります．2番目の命令の「fetch」ステージが1番目の分岐先の計算の影響で遅れるからです．悪いことに，2番目の命令のフェッチは，分岐命令がデコード中でまだ分岐だと認識されていないときに起こるため，フェッチされた命令が捨てられてしまうこともあります．たとえば，分岐ターゲット計算が図1.13のパイプラインの「ALU」ステージで起こったとすると，分岐先が確定する前に，古いストリームから3命令がフェッチされてしまっていることになります．

専用ハードウェアが必要になるにしても，できれば分岐先をパイプラインの早期に計算したほうが効率的です．分岐命令が固定した形式をもつ場合，分岐先は「dec」ステージの間に**投機的**（命令が分岐だと認識される以前に）に割り出せます．そうすれば，分岐の待ち時間は1サイクルになりますが，条件的分岐の場合は分岐を先行する命令の条件コード結果に依存しているため，このパイプラインではまだ障害が残ります．RISCアーキテクチャ（ARMではありません）の中には，分岐をするかどうかにかかわらず，次の命令を実行するもの

もあります．これは**遅延分岐**(delayed branch)と呼ばれます．

◆パイプライン効率

上記のパイプライン障害の影響を減らす方法はありますが，完全な解決はできません．パイプラインが深いほど(ステージが多いほど)問題は悪化します．きわめて単純なプロセッサでは，3～5ステージのパイプライン導入は大きな利点があります．しかし，それ以上になると，収益てい減の法則が効いてきてコストと複雑さの増大が利点を上回ります．

すべての命令が似たようなシーケンスをたどる場合は，パイプラインは明らかに効果があります．各命令がどれも異なる動きをするような複雑な命令セットをもったプロセッサは，パイプラインに向きません．1980年代当時のCISCプロセッサは，シリコン・リソース不足，設計リソース不足，複雑な命令セット向けのパイプライン設計の複雑さなどの理由で，パイプライン化されていませんでした．

1.6　縮小命令セット・コンピュータ(RISC)

1980年，"The Case for the Reduced Instruction Set Computer(縮小命令セット・コンピュータの研究)"と題した論文がPattersonとDitzelによって発表されました(参考文献リストに詳細情報がある)．2人はこのゼミ研究で，シングルチップ・プロセッサに最適なアーキテクチャは必ずしもマルチチップ・プロセッサに最適なアーキテクチャと同一である必要はない，という考えを披露しました．そして，この議論はカリフォルニア大学バークレー校の大学院生によるプロセッサ設計プロジェクトの結果によって裏付けられました．そのプロジェクトで採用されたのが縮小命令セット・コンピュータ(RISC)です．設計されたバークレーRISC I は，当時市販されていたCISCプロセッサよりずっと単純で，設計開発の労力もかなり少なくてすみましたが，それでいて同じくらいの性能を発揮しました．

RISC I 命令セットは，市販のマイクロプロセッサに使用されていたミニコンピュータのようなCISC命令セットとは，多くの点で異なっていました．RISC Iのおもな特徴は次のとおりでした．

◆RISCアーキテクチャ

(1) ほとんど同じフォーマットをもった固定長(32ビット)命令サイズ．CISCプロセッサの命令セットは，一般的に長さは一定せず，多くのフォーマットが存在しました．

(2) ロード/ストア・アーキテクチャにより，データを処理する命令セットがレジスタでのみ処理され，メモリにアクセスする命令とは区別されていること．CISCプロセッサはメモリ内の値をデータ処理命令のオペランドとして使用できました．

(3) 32本の32ビット・レジスタという大きなレジスタ・バンク．全レジスタがどんな目的にも使えるため，ロード/ストア・アーキテクチャの動作が効率的に実行される．CISCレジスタ・セットも大きくなっていたが，これほどではなく，多くは目的ごとに異なるレジスタをもっていた(たとえば，Motorola社68000のデータ・レジスタとアドレス・レジスタ)．

上記の違いによって，プロセッサ設計が大幅に単純化されたため，設計者はプロトタイプ・デバイスの性能を向上するような構成上の特徴をもつアーキテクチャを実現できるようになりました．

◆RISC構成

(1) ハード・ワイヤード命令デコード論理．CISCプロセッサは命令のデコードに大きなマイクロコードROMを使っていた．

(2) パイプライン処理．CISCプロセッサは，連続する命令の実行がオーバラップをほとんど許さないか，やってもわずかである(今はそうでもない)．

(3) 単一サイクル処理．CISCプロセッサは，一つの命令を実行するのに多くのクロック・サイクルを使っていた．

上記のアーキテクチャ上と，構成上の変更をすべて組み込むことで，バークレーのRISCマイクロプロセッサは，段階的な改善では対処できなかった問題を解決しました．また，部分的にパフォーマンスを最大にしても，全体のパフォーマンスが最大になるかわからないという問題から解放されました．

◆RISCの利点

PattersonとDitzelは，RISCには，次の三つの大きな利点があると主張しました．

(1) ダイ・サイズの縮小

RISCプロセッサは，少ないトランジスタとシリコン面積で実装できます．したがって，高度なプロセス技術を使用せずとも，CPU全体を1個のチップ上に収めることができます．たとえCISCプロセッサでも実装できるようなプロセス技術を用いたとしても，RISC CPUのほうがより広い空きダイ領域をもつことができ，この空きダイ領域をキャッシュ・メモリ，メモリ管理機能，浮動小数点ハードウェアなど性能を向上する機能に使えるようになりました．

(2) 開発時間の短縮

RISCプロセッサは設計にかける労力やコストが少なくてすみ，製品化時点でのプロセス技術によりよく適合するでしょう(より短い開発期間の間のプロセス技術の進歩を，予測すればよいからである)．

(3) 高性能化

これは，ややわかりにくいところです．前述の二つの利点はすぐ理解できますが，複雑さを増し続けることで高性能を追い求めていた世界において，これは納得しがたいかもしれません．

理屈はこうです．小さいものはより高い固有周波数をもつので(昆虫は小鳥より速く羽根を動かすし，小鳥は大型の鳥よりも速く羽根を動かす)，単純なプロセッサはクロック速度が速いはずです．単純なプロセッサから設計をはじめ，徐々に複雑な命令を足して複雑なプロセッサを設計してみましょう．複雑な命令を足すと，高級機能をより効率的に処理しますが，全命令に関してクロック速度を多少落としてしまいます．代表的なプログラムについて全体での効果を測定してみると，複雑な命令はどれもプログラムの実行速度を落とすのです．したがって，プロセッサは単純なほうが性能を向上しやすいということになります．

上記の理論は，実験結果とプロトタイプ・プロセッサ(バークレーRISC II もRISC I に続いて完成した)によって裏付けられました．商用的なプロセッサ・メーカは当初懐疑的でしたが，独自の目的でプロセッサを設計している新しい会社の多くは，開発コストを減らしてライバル社に勝つチャンスだと考えました．ARMがトップを切った商用的RISC設計は，RISCの発想が正しかったことを証明しました．1980年以降，新しい汎用プロセッサのアーキテクチャはどれも多かれ少なかれRISCの概念に基づいています．

◆RISCを振り返る

RISCは今や商用的に確立されています．ここでRISCがマイクロプロセッサの進化にどう貢献したのかを具体的に見てみましょう．初期のRISCは，次の方法で高性能を達成しました．

(1) パイプライン

パイプラインは，プロセッサ内に実装されるもっとも簡単な方法は並列化であり，これにより並列処理を実現するもっとも単純な方法で，2～3倍に高速化します．単純な命令セットはパイプライン設計も大幅に単純化します．

(2) 単一サイクル処理による高クロック速度

1980年当時，標準的半導体メモリ（DRAM）は約3MHzのランダム・アクセス，6MHzのシーケンシャル（ページ・モード）アクセスが可能でした．当時のCISCマイクロプロセッサは，最大2MHzでしかメモリにアクセスできず，メモリの帯域幅を完全に活用していませんでした．RISCプロセッサは単純でしたが，使用可能な帯域幅をフルに使うクロック速度で動作するよう設計されていました．

上記のものは，いずれもアーキテクチャ上の特徴ではありませんが，どちらもアーキテクチャがそれを組み込めるだけ単純である必要があります．RISCアーキテクチャが成功したのは，単純なために設計者がこれらの構成テクニックを活用できたからです．マイクロコードを使った固定長命令ロード/ストア・アーキテクチャ，マルチサイクル実行，パイプラインなしを実装することはもちろん可能でしたが，そのような実装なら市販のCISCを使っても同じことです．当時，ハード・ワイヤード，単一サイクル実行，パイプライン方式のCISCを実現することは不可能だったのです．ただし現在は，可能です．

◆クロック速度

上記の分析に加え，クロック速度についてはさらに説明すべき二つの側面があります．

(1) 1980年代のCISCプロセッサは，初期のRISCよりも速いクロック速度で動作しましたが，1回のメモリ・アクセスに数クロック・サイクルかかっていたので，メモリ・アクセス速度は遅いものでした．クロック速度だけでプロセッサを評価するのは禁物です！

(2) CISCのメモリ・アクセス速度とメモリ帯域幅のずれは，1.5節の「複雑命令セット・コンピュータ」の項で述べたことと矛盾するように思えるかもしれません．そこでは，プロセッサ速度に比例して主メモリ速度が遅いことを根拠として，1970年代初期のミニコンピュータでマイクロコードを使用することを肯定しているからです．この矛盾は，その10年間でメモリ技術が非常に高速化した一方で，初期のCISCマイクロプロセッサが一般的なミニコンピュータ・プロセッサよりも遅かったことを考えると納得できます．プロセッサ速度が遅い原因は，完全なプロセッサを1個のチップに載せるのに必要な回路密度を得るためには，高速のバイポーラ技術から遅いNMOS技術に移行しなければならなかったからです．

◆RISCの弱点

RISCは性能において明らかに優れ，設計コストも安価です．それでは，RISCはいいことばかりなのでしょうか？　時間がたつにつれ，二つの弱点が明らかになりました．

(1) CISCに比べ，一般的にコード密度が低い．

(2) x86コードが実行できない．

このうち2番目の問題の解決は困難です．ただし，これはIBM PC互換機を作ろうとする場合だけです．他のアプリケーションに関してはまったく無視できます．

固定長の命令セットに起因する低いコード密度は，より多くのアプリケーションにあてはまる深刻な問題です．キャッシュなしの場合，コード密度が低いとメイン・メモリの帯域幅が，より命令フェッチのために使われることになり，メモリの消費電力が大きくなります．プロセッサが一定のサイズのオンチップ・キャッシュを組み込んでいる場合，コード密度が低いと，つねにキャッシュには動作に必要な命令の一部分しか入ってい

ないことになり，キャッシュのミス率が高くなり，結果としてかえって広いメイン・メモリ帯域幅が必要になり消費電力も増えます．

◆ARMコード密度とThumb

ARMプロセッサの設計はRISCの原理に基づいていますが，多くの工夫のおかげで低コード密度による悪影響が他のRISCより少なくなっています．それでも，コード密度はCISCプロセッサほどよくありません．コード密度を最優先にしたい場合のために，ARM社はThumbアーキテクチャを用意し，ARMプロセッサの一部のバージョンに組み込みました．

Thumb命令セットは，元の32ビットARM命令セットを16ビットに圧縮したもので，命令パイプライン内に動的伸張ハードウェアを採用しています．Thumbのコード密度は，ほとんどのCISCプロセッサよりも優れています．

Thumbアーキテクチャについては，第7章で解説します．

◆RISCを超えて

コンピュータ・アーキテクチャの進歩がRISCで終わるはずはありません．RISCが時代遅れになるような新たな発明の兆候はあるのでしょうか？

これを書いている時点で，RISCと同程度の変化を引き起こしそうな開発は見当たりません．しかし，命令セットは進化を続けており，より効率的な実装やマルチメディアのような新しいアプリケーションにも対応するようになっています．

1.7　低消費電力を目指した設計

ディジタル・コンピュータが導入されてから50年，コスト効率における向上は，他のいかなる技術分野よりも目覚ましいものがあります．性能向上と同時に，マシンの消費電力は著しく低減されました．もっとも最近では，性能を向上させるのと同じくらい消費電力を下げることが重要になってきています．いくつかのアプリケーション領域ではとくに重要です．これは，ディジタル携帯電話やラップトップ・コンピュータなど，高性能の処理コンポーネントを組み込んだ電池で動く携帯機器の市場が伸びている結果です．

ICの導入につれて，コンピュータ業界はトランジスタの小型化と低コスト化，性能向上と消費電力低減という相乗効果によって，発展の一途をたどってきました．しかし，今や設計者は，場合によっては性能を犠牲にしても，消費電力低減に焦点を絞って設計するようになりました．

ARMプロセッサは，この電力効率の良いプロセッサを求める流れの中心です．ここで，低消費電力を目指した設計に関わる問題を見てみましょう．

◆電力のゆくえ

低消費電力設計の出発点は，電力が回路のどこに消えるのかを理解することです．CMOSは現代の高性能ディジタル・エレクトロニクスにおける主要技術で，低消費電力設計に関しても良い性質をもっています．ここで，CMOS回路において電力がどこへ行くのかを見てみましょう．

典型的なCMOS回路は，前に示した図1.2にあるようにスタティックNANDゲートです．すべての信号は電源電圧と接地の間，つまりV_{dd}とV_{ss}の間を行き来します．つい最近まで，5Vの電源が標準的でしたが，現代の

CMOSプロセスは3V程度の低電圧しか必要としませんし，最新の技術では，1Vないし2Vの電圧で動作します．将来はさらに下がると考えられます．

ゲートは，出力をV_{dd}へp型トランジスタのプルアップ・ネットワークを通じてつなぐか，V_{SS}へn型トランジスタのプルダウン・ネットワークを通じてつなぐかのどちらかによって動作します．両方の入力がどちらかの電圧に近い場合，どちらかのネットワークが導通し，もう片方が導通しませんから，V_{dd}からV_{SS}へゲートを通じて行く道はありません．さらに，出力は普通，同様のゲートの入力につながっていますから，容量負荷しかありません．いったん出力がどちらかの電圧に近くなると，電力は必要なくなります．したがって，ゲートがスイッチ動作するとすぐ，回路は安定した状態になり，それ以上電源から電流は流れません．

スイッチするときだけ電力を消費するという性質は，他の回路技術にはまれです．このため，CMOSが高集積度ICに最適な技術と言われているのです．

◆CMOSの電力要素

CMOS回路の総電力消費は，三つの要素から成っています．

(1) スイッチ電力

ゲート出力静電容量C_Lの充電，放電で消費するもので，ゲートによる有用な仕事量を表します．

出力遷移ごとのエネルギは，次のように表せます．

$$E_t = (1/2)\, C_L \cdot V_{dd}^2 \qquad \text{（式 2）}$$
$$\fallingdotseq 1 \text{（ピコ・ジュール）}$$

(2) ショート電力

ゲート入力が中間レベルのときは，pとn両タイプのネットワークが導通します．このため，V_{dd}からV_{SS}へ一時的な導通経路を作ります．正しく設計された回路なら（通常，信号遷移が遅くないもの），ショート電力はスイッチング電力のごく一部を占めるにすぎません．

(3) 漏れ電流

トランジスタは，オフ状態でも微量の電流を通します．一般的なプロセスにおいてこの電流は微量（1ゲートにつき，1nAの何分の1か）ですが，通電中でかつ非アクティブな回路では，これが唯一の電力損失です．また，長期間には電源バッテリの漏れにもつながります．アクティブな回路においては，通常無視できます．

正しく設計された動作の回路ではスイッチング電力が優勢であり，ショート電力は総電力を10～20%増すだけ，また漏れ電流は回路が非動作の場合にのみ関係してきます．しかし，後で詳しく述べるように，低電圧動作の傾向が進むにつれて，性能と漏れ電流の間にトレードオフが起こっています．そして漏れ電流は，将来の省電力高性能設計にとって大きな問題になると思われます．

◆CMOS回路電力

CMOS回路の総電力消費P_Cは，ショートと漏れを無視すると，回路Cの各ゲートgの電力消費を合計して求めることができます．

$$P_C = (1/2) f \cdot V_{dd}^2 \sum_{g \in C} A_g \cdot C_L^g \qquad \text{（式 3）}$$

fはクロック周波数，A_gはゲートの動作率（全ゲートがクロック・サイクルごとにスイッチするわけではないので），C_Lはゲート負荷容量です．クロック・ラインを合計するうえで，クロック・サイクルごとに2回遷移

するので，動作率は2となることに注意してください．

◆低消費電力回路設計

一般的ゲートの負荷容量はプロセス技術の範ちゅうであり，設計者には，直接かかわりがありません．(式3)の残りのパラメータにより様々な低消費電力設計へのアプローチがあることがわかります．次に，それらを重要な順に挙げます．

1）電源電圧V_{dd}を最小化する

消費電力に対して電源電圧の2乗で効くので有力な候補です．後で詳しく検討します．

2）回路の動作率Aを最小化する

ゲート付きのクロックのような技術もこれに含まれます．回路が機能する必要のないときは，つねに動作を止めるべきです．

3）ゲート数を減らす

ほかの条件がすべて同じなら，単純な回路は複雑なものより消費電力が少なくてすみます．なぜなら，電力消費に結びつくゲートの合計数が少ないからです．

4）クロック周波数fを最小化する

不必要にクロック速度を高くすることを避けるのはもちろんです．低いクロック速度は消費電力を減らしますが，性能も落ちるため，電力効率の点では効果がありません(たとえば，1WあたりのMIPSで測るとわかる)．しかし，クロック周波数を落とせば低いV_{dd}で動作できるので，電力効率の点で多いに効果が上がります．

◆V_{dd}の低減

CMOSゲートの形状が小さくなるにつれ，電源電圧も低減しなければなりません．なぜなら，トランジスタを作っている物質は強い電界には耐えられないのですが，電源電圧が一定の場合，トランジスタが小さくなればなるほど電界が強くなるからです．

しかし，低消費電力を優先した設計への要求が高まってくると，たんに電気的故障を防ぐためではなく，電源電圧を下げることが求められるようになりました．電源電圧の低減を妨げる要因はなんでしょう？

問題は，V_{dd}を低減すると回路の性能も落ちてしまうことです．飽和トランジスタ電流は，次のように求められます．

$$I_{sat} \propto (V_{dd} - V_t)^2 \qquad \text{（式 4）}$$

V_tはトランジスタのしきい値電圧です．回路ノードにおける電荷は，V_{dd}に比例しますから，最高動作周波数は次のように求められます．

$$f_{max} \propto (V_{dd} - V_t)^2 / V_{dd} \qquad \text{（式 5）}$$

したがって，電源電圧V_{dd}が低くなれば，最高動作周波数も低くなります．高電圧での電流は速度飽和効果による限界があるため，サブミクロン・プロセスにおける性能低下は，おそらく(式5)に見えるほど深刻ではありません．しかし，性能はある程度落ちます．(式5)を見ると，性能低下を抑える確実な方法は，V_tを下げることです．しかし次に示すように，漏れ電流はV_tに大きく影響されます．

$$I_{leak} = exp(-V_t / 35\text{mV}) \qquad \text{（式 6）}$$

V_tが少し減っただけでも，漏れ電流はかなり増加し，動作していない回路からの電流漏れが増加します．し

たがって，性能の最大化とスタンバイ電力の最小化との間でのトレードオフが問題になります．両方の特性が重要である場合，システム設計者はこの問題を慎重に考慮しなければなりません．

たとえスタンバイ電力が重要でない場合でも，非常に低いしきい値のトランジスタを使って性能を最大化すると，漏れ電力が動的な動作に匹敵するほど大きくなるおそれがあることに設計者は注意しなければなりません．ですから，パッケージと冷却システムを選択する際には，漏れ電力に考慮することが必要です．

◆低消費電力への対策

低消費電力を目指す設計技術について基本事項を述べてきましたが，締めくくりの前に低消費電力アプリケーションを作るための対策をあげてみます．

(1) V_{dd}の最小化

必要な性能を出せる最低のクロック周波数を選択し，電源電圧をクロック周波数やシステム・コンポーネントの条件内で最低に設定します．漏れ電流とスタンバイ電力の釣り合いがとれる程度に供給電圧を下げるよう注意してください．

(2) オフチップでの動作の最小化

オフチップの容量は，オンチップの負荷容量よりずっと大きいので，オフチップの動作はつねに最小にしなければなりません．オフチップ負荷を駆動するための信号遷移をできるだけ避けるようにします．たとえば，オフチップ・メモリへのアクセスを最小にするためキャッシュを用いたりします．

(3) 並行処理の利用

電源電圧が自由ならば，電力効率を上げるために並行処理を利用します．回路を二重にすれば，元の回路の半分のクロック周波数で2回路が同じ性能を果たします．したがって，低い電源電圧で必要な性能が可能になります．

低消費電力設計については，盛んに研究がなされており，新しいアイデアが次々に生まれています．向上するプロセス技術と設計技術を組み合わせることにより，今後5～10年の間に，高速ディジタル回路の電力効率向上において，画期的進歩があるのではないかと期待されています．

1.8　例題と演習

(実際に演習を解くには，ハードウェア・シミュレーション環境へのアクセスが必要になります．)

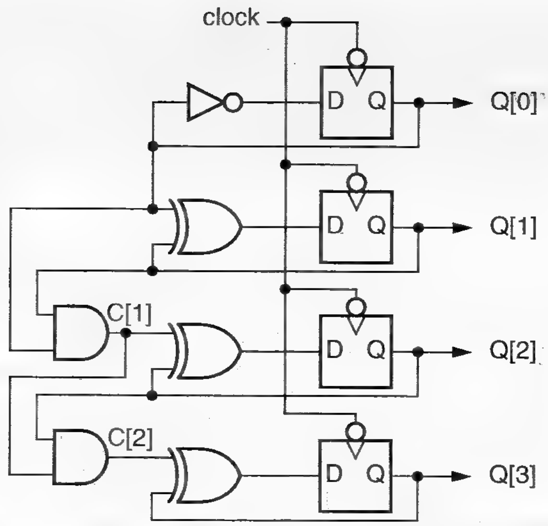
＜例題1.1＞　論理ゲートと4ビット・レジスタを使った4ビット・バイナリ・カウンタを設計しなさい．

レジスタ入力を$D[0]$から$D[3]$で表し，出力を$Q[0]$から$Q[3]$で表した場合，カウンタは，$D[3:0]=Q[3:0]+1$を生成する組み合わせ論理を構築することで実装します．バイナリ加算器の論理方程式は付録A〔(式18)で和を求め，(式19)で桁上げを求める〕にあります．第2オペランドが一定ならば，これらの方程式は次のように単純化できます．

$$D[0] = \overline{Q[0]} \qquad \text{(式7)}$$

$$D[i] = Q[i] \cdot \overline{C[i-1]} + \overline{Q[i]} \cdot C[i-1] \qquad \text{(式8)}$$

$$C[i] = Q[i] \cdot C[i-1] \qquad \text{(式9)}$$



〔図1.16〕 4ビット・バイナリ・カウンタの論理回路

ただし，$1 \leqq i \leqq 3$，かつ$C[0]=1$とする（$C[3]$は不要）.

これらの式は図1.16のように，レジスタも含む論理回路として描けます.

演習1.1.1　バイナリ・カウンタを変更して0-9をカウントするようにし，次のクロック・エッジで再び0から開始するようにします（これは**モジュロ10**カウンタと呼ぶ）.

演習1.1.2　**同期クリア**機能をもつようにバイナリ・カウンタを変更します．つまり，新しい入力（クリア）を与えると，アクティブなら現在の値が何であれ次のクロック・エッジの後はカウンタ出力が0になります.

演習1.1.3　**アップ/ダウン**入力をもつようにバイナリ・カウンタを変更します．この入力が“H”のとき，カウンタは上の例で説明したように動作し，“L”のときはカウンタがカウント・ダウンします（アップ・モードの逆順序で）.

<例題1.2>　MUO命令セットにインデクス・アドレシングを追加しなさい.

ここで有効な最小限の拡張は，新しい12ビットのインデクス・レジスタ（X）と，それを初期化し命令をロード/ストアできるようにする新しい命令を導入することです．前述の**表1.1**を見ると，当初の設計中で八つのOPコードしか使われていません．ですから，新しい命令を八つ加える空間が残っていることになります．基本的な一連のインデクス操作は次のとおりです.

```
LDX  S          ; X := mem16[S]
LDA  S, X       ; ACC := mem16[S+X]
STA  S, X       ; mem16[S+X] := ACC
```

もし，インデクス・レジスタを修飾する何らかの方法があれば，もっと便利です．たとえば次の表を使う場合です．

```
INX              ; X := X + 1
DEX              ; X := X − 1
```

ここでは，インデクス・レジスタの基本的機能性が示されています．インデクス・レジスタをメモリにストアする方法があると，テンポラリ(一時的な)レジスタとして使うことができるので，便利になります．しかし，これ以上は話が複雑になるのでここで止めます．

演習1.2.1 必要なら新しい制御信号を追加して，図1.6のようにXレジスタを含むようにRTL構成を変更しなさい．

演習1.2.2 表1.2のように制御論理を変更し，インデクス・アドレシングに対応しなさい．ハードウェア・シミュレータにアクセスできるなら自分の設計をテストしてみてください(これは大変な仕事です！)．

<例題1.3> 単一サイクル遅延分岐による性能向上を予測しなさい．

遅延分岐により，分岐をするか否かにかかわらず，分岐に続く命令が実行できます．分岐に続く命令は「遅延スロット」に入っています．動的な命令頻度は表1.3，パイプライン構造は図1.13であると仮定します．レジスタ障害は無視し，遅延スロット全部が埋まると仮定します(ほとんどの単一遅延スロットは埋まる)．

デコード・ステージに専用の分岐ターゲット加算器がある場合は，分岐は1サイクル遅延効果があり，単一遅延スロットが無駄になったサイクルを全部取り除きます．命令の四つに一つは分岐ですから，遅延スロットがなければ4命令に5クロック・サイクル使うことになり，4クロック・サイクルですめば25%も性能が向上します．

専用の分岐ターゲット加算器がなく，主ALUステージがターゲット計算に使われている場合は，一つの分岐で3サイクルを無駄にします．したがって，平均1分岐を含む4命令は7クロック・サイクルを使い，遅延スロットが一つある場合は6クロック・サイクル使います．つまり遅延スロットは17%性能を向上します(しかし，遅延スロットなしでも専用分岐加算器は有効である)．

演習1.3.1 2サイクル遅延分岐による性能向上を予測しなさい．最初の遅延スロットは全部埋まり，第2の遅延スロットは50%が埋まると仮定します．

なぜ，2サイクル遅延分岐は専用分岐ターゲット加算器がない場合にのみ意味があるのでしょう？

演習1.3.2 上記の1サイクル，2サイクル遅延分岐がコード・サイズに与える影響は何でしょう？(埋まっていない分岐遅延スロットはすべてNOPで埋めなければならない)

第2章 ARMアーキテクチャ

ARMプロセッサは，縮小命令セット・コンピュータ(RISC)です．前章で示したように，RISC概念は，1980年ごろに，スタンフォード大学，カリフォルニア大学バークレー校の研究課程で生まれました．

本章では，RISC思想がARMプロセッサ誕生にどう貢献したかを見ます．ARMは，当初(1983-1985年ごろ)英国ケンブリッジ市のAcorn Computers Limitedで開発されました．それは商用に開発された最初のRISCマイクロプロセッサで，その後のRISCアーキテクチャとは明らかに異なっている点がいくつかあります．ここでは，ARMアーキテクチャの主要な特徴をまとめることにし，詳細は後の章で説明します．

1990年，Advanced RISC Machines Limitedは，ARMテクノロジの利用を拡大する目的で設立されました．以来，ARMは世界中の多くの半導体メーカにライセンスされています．そして，低消費電力でコスト重視の組み込み型アプリケーションの市場リーダとしての地位を確立しました．

ハードウェアおよびソフトウェア開発ツールなしでは，どんなプロセッサも有効に活用できません．ARMをサポートするツールキットには，ハードウェア・モデリング，ソフトウェア・テスト，ベンチマーク，アセンブラ，CおよびC++コンパイラ，リンカやシンボリック・デバッガが含まれます．

2.1 Acorn RISCマシン

最初のARMプロセッサは，1983年10月から1985年4月の間に，英国ケンブリッジ市のAcorn Computers(エイコーン・コンピュータ)社で開発されました．当時(Advanced RISC Machines 社が1990年に設立されるまで．後に単純なARM Limited と改名されることになる)，ARMは**Acorn RISC**マシンと呼ばれていました．

Acornは，BBCマイクロコンピュータの成功によって，英国のパソコン市場で優位に立ちました．BBCマイクロは，8ビットの6502マイクロプロセッサを搭載したマシンで，1982年1月以来，BBCが放送したテレビ・シリーズの影響で英国の学校に広く普及しました．BBCマイクロは，ホビーイストにも熱狂的に支持され，研究所や高等教育施設にも導入されました．

BBCマイクロの成功の後，Acornのエンジニアは次のマシンを作るために多くのマイクロプロセッサのなかから探そうとしましたが，どの商用プロセッサも物足りませんでした．1983年当時出まわっていた16ビットCISCマイクロプロセッサは，標準的メモリ・チップより低速でした．また，多くのクロック・サイクル(ときには，数百クロック・サイクル)を必要とする命令セットを使っていたため，割り込み遅延時間がとても長く

なっていました．BBCマイクロは6502の高速割り込み応答の特徴を最大限に生かしていたので，Acornの設計者はプロセッサの性能で後戻りとなるような手段を選ぶ気にはなれませんでした．

市場に出ていたマイクロプロセッサに対する不満から，Acornは独自のマイクロプロセッサの設計を考えました．問題は，商用マイクロプロセッサ開発には非常に長い年月と設計労力がかかることです．Acornは総従業員数がやっと400人を超える会社でしたから，そのような規模の投資は無理でした．BBCマイクロ用に設計した数個の小型ゲートアレイ以外にはカスタムチップ設計の経験もまったくありませんでした．そんな状況の中，少ない設計労力で良い設計を生み出さなければならなかったのです．

こんな危機的状況に，突然現れたのがバークレーのRISC Ⅰの論文でした．数人の大学院生が1年足らずで設計し，それでいて市場トップの製品に劣らないプロセッサです．そもそも単純でしたから，割り込み応答の遅れ時間を延ばすような複雑な命令は用いていません．将来の方向性を示唆しているという評判でしたが，技術的メリットがいかに学術的に評価されていようとも，商業的成功に結びつくという保証はありませんでした．

こうして，偶然の重なりによってARMは，Acornの製品ラインの核となりました．その後，**Advanced RISC Machine**と名称を変え，さらにAcorn製品の域を超えて市場を広げるため，会社の名称ともなりました．名称は変わりましたが，アーキテクチャは元のAcornでの設計をほとんど踏襲しています．

2.2 アーキテクチャの継承

初代のARMチップが設計された当時，RISCアーキテクチャはバークレーRISC Ⅰ，Ⅱ，スタンフォードMIPS（“Microprocessor without Interlocking Pipeline Stages”の略）しか例がありませんでした．もっとも，RISCに先立つディジタルPDP-8，Cray-1，IBM 801なども，後にRISCと関連づけられる性質を多く備えていました．

◆バークレーRISCから引き継がれた特徴

ARMアーキテクチャは，バークレーRISCの設計から多くの特徴を採用しましたが，採用しなかった特徴も多くありました．採用したのは次のとおりです．

(1) ロード/ストア・アーキテクチャ
(2) 固定長32ビット命令
(3) 3アドレス命令形式

◆バークレーRISCから引き継がれなかった特徴

バークレーRISCの設計にはありましたが，ARMの設計に採用されなかった特徴は次のとおりです．

(1) レジスタ・ウィンドウ

バークレーRISCプロセッサのレジスタ・バンクは多数のレジスタを組み込んでいて，うち32本のレジスタがつねに可視でした．プロシージャ・エントリおよびエグジット命令が可視「ウィンドウ」を動かして各プロシージャを新しいレジスタにアクセスさせ，それによって，レジスタ保存と復元に起因するプロセッサとメモリ間のデータ・トラフィックを軽減させていました．

レジスタ・ウィンドウの大きな問題は，多数のレジスタが占めるチップ面積です．そのためARMの設計ではコスト面から採用されませんでしたが，ARMで例外を処理するシャドウ・レジスタは概念上，これに似ています．

RISCの初期，このメカニズムは，バークレー・プロトタイプに含まれていたためRISC概念と非常に強く結びついていました．しかしその後，当初の形で採用したのはSun SPARCアーキテクチャのみです．

2）遅延分岐

分岐は命令のスムーズな流れを妨害するため，パイプライン上の問題を生みます．多くのRISCプロセッサは分岐命令以降の命令を実行したあとで分岐を実行する遅延分岐を使うことで障害を軽減しています．

遅延分岐の問題点は，個々の命令が不可分でなくなることです．遅延分岐は単一パイプライン・プロセッサではうまく機能しますが，スーパースカラでの実装ではうまく機能しません．また，分岐予測メカニズムとの相性もよくありません．

当初のARMでは，遅延分岐は例外処理を複雑化するという理由で用いられていませんでした．その後，このことは，異なるパイプラインでアーキテクチャの再実装を単純にすることがわかり，良い決断だったということになりました．

3）全命令の1サイクル実行

ARMのデータ処理命令のほとんどを1クロック・サイクルで実行します．その他の命令は複数のクロック・サイクルが必要です．

この理由は，一つのメモリにデータと命令がともに存在すると，単純なロードやストア命令でもメモリ・アクセスを最低2回使ってしまうからです（命令に1回，データに1回）．したがって，全命令の1サイクル実行は，データと命令のメモリが分かれている場合のみ可能であり，ARMが意図するアプリケーションには高価すぎると考えられました．

全命令の1サイクル化の代わりに，ARMはメモリ・アクセスに必要なサイクル数を最小化するよう設計しました．1サイクルより多いときは，よけいにかかるサイクルは可能ならば自動インデクス・アドレシング・モード対応など有効に使うようにします．これによって性能とコード密度を向上し，各処理シーケンスの実行に必要なARM命令数を低減できます．

◆簡潔さ

当初のARM設計チームが一番気を使ったのは，設計を単純にすることでした．初代ARMチップ以前，Acornの設計者は約2000ゲートのゲートアレイしか設計した経験がありませんでしたから，フルカスタムCMOSの設計には気を使いました．未知の領域に足を踏み入れる場合は，制御できるリスクは極力抑えることです．それでもなお，十分に理解できていない要素や，制御不可能な要素から起こるリスクが多数あるからです．

ARMの単純さは，命令セット・アーキテクチャよりもハードウェア構成および実装を見れば（第4章で説明する）はっきりわかります．これはプログラマの視点からすると，RISCの指針を基本的に受け入れながらも，後続のRISC設計ほど革新的とはいえないARM命令セット設計の保守性が見えています．

ARM命令セットは，RISC概念に基づきながらも主要なCISCの特徴を残すことにより，純粋なRISCよりも優れたコード密度を達成しました．単純なハードウェアとこの命令セットを組み合わせることにより，ARMの電力効率と小さなコア・サイズが生まれたのです．

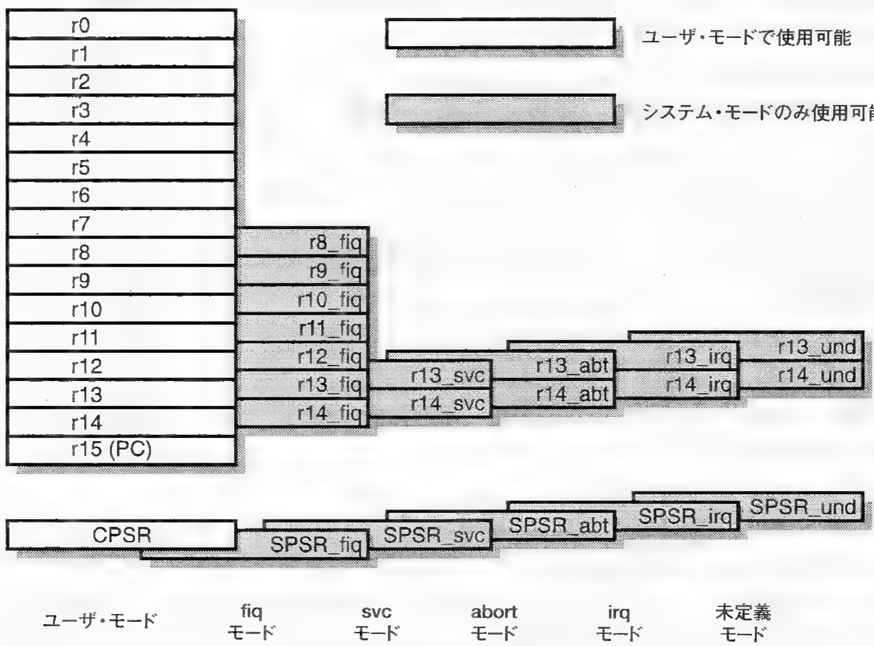
2.3　ARMプログラマ・モデル

プロセッサの命令セットは処理を定義し，プログラマがプロセッサを組み込んでいるシステムの状態を変化

させるのにこの命令セットを用います．システムの状態とは，プロセッサの可視レジスタおよびシステム・メモリ内のデータ項目の値で構成されます．

各命令は，命令が実行される前の状態から命令実行後の状態への変化を決められたとおり実行すると考えることができます．プロセッサは，通常，命令の実行に関わる多くの不可視メモリをもっていますが，これらレジスタの命令実行前後の値は重要ではありません．可視レジスタの値だけが重要なのです．ARMプロセッサの可視レジスタは，図2.1のとおりです．

ユーザ・レベルのプログラムを書く際は，15本の汎用32ビット・レジスタ（r0からr14），プログラム・カウンタ（r15），カレント・プログラム・ステータス・レジスタ（CPSR）だけを考えます．残りのレジスタは，システム・レベル・プログラミングと例外処理（割り込みなど）にしか使いません．



〔図2.1〕ARMの可視レジスタ



〔図2.2〕CPSRのフォーマット

◆カレント・プログラム・ステータス・レジスタ(CPSR)

CPSRはユーザ・レベルのプログラムに使用され，条件コード・ビットを保存します．これらのビットは，たとえば比較の結果を記録したり，条件分岐を実行するかどうかを制御します．ユーザ・レベルのプログラマは，基本的にはレジスタの構成に気を配る必要はありませんが，念のため図2.2にレジスタを図示します．

レジスタの構成を下位ビットから見ると，プロセッサ・モード“mode”（5.1節を参照），命令セット‘T’（7.1節を参照），割り込みイネーブル‘I’および‘F’（5.2節を参照）を制御し，ユーザ・レベル・プログラムによる変更から保護されています．条件コード・フラグはレジスタの最上位4ビットにあり，次の意味をもっています．

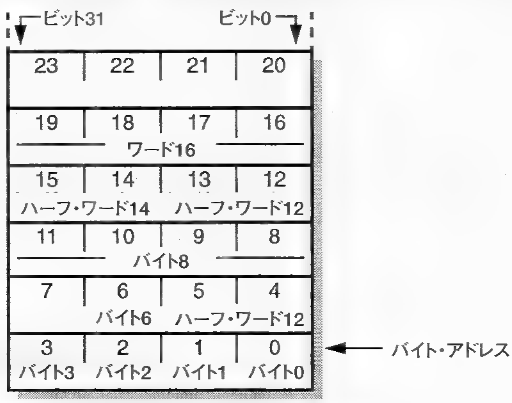
- 'N'：負(Negative)．フラグを変更した最後のALU処理が，負の結果を生成(32ビット結果の最上位ビットが'1')
- 'Z'：零(Zero)．フラグを変更した最後のALU処理が，零の結果を生成(32ビット結果の全ビットが'0')
- 'C'：桁上げ(Carry)．フラグを変更した最後のALU処理が，ALUの算術計算またはシフタにより桁上げ出力(キャリアウト)を生成
- 'V'：あふれ(oVerflow)．フラグを変更した最後のALU処理が符号ビットにあふれを生成

上記のうちCとVの定義はややこしく見えますが，それらの処理を詳しく理解しなくても使用はできます．たいていの場合，簡単な条件テストがあり，プログラマが条件コード・ビットの正確な値を計算しなくても必要な結果が出ます．

◆メモリ・システム

プロセッサ・レジスタ状態に加えて，ARMシステムにはメモリ状態があります．メモリは0から$2^{32}-1$までのバイトの線形配列と考えられます．データ項目は，8ビット＝バイト，32ビット＝ワード，また一部のARMシステムでは16ビット＝ハーフ・ワードとなります．各ワードはつねに4バイトの境界をもって並んでおり(つまり，最下位二つのアドレス・ビットは0)，ハーフ・ワードは偶数バイトの境界をもって並んでいます．

メモリ構成は図2.3に示しました．これは，各バイトの位置が独自の番号をもっている小さなメモリ領域からなります．バイトはこれらのどの位置を占めることもできますが，いくつかの例が図に示してあります．ワード・サイズのデータ項目は，4の倍数のバイト・アドレスから始まる一連の4バイト位置を占めます．この例も図に示しました．ハーフ・ワードがサポートされている場合は，偶数バイト・アドレスで始まる2バイト位置を占めます(これが，標準的な「リトル・エンディアン」，つまりARMが使用しているメモリ構成である．ARMは「ビッグ・エンディアン」メモリ構成で動作するようにも設定できる．詳しくは第5章で述べる)．



〔図2.3〕ARMメモリ構成

◆ロード/ストア・アーキテクチャ

多くのRISCプロセッサと同様，ARMはロード/ストア・アーキテクチャを採用しています．つまり，命令セットは，レジスタにある値(もしくは，命令そのものの中で直接指定されているもの)しか処理(加算，減算など)しないということです．そして，処理結果はつねにレジスタに保存されます．メモリに対して適用される唯一の操作は，「メモリ値をレジスタにコピーする(ロード命令)こと」と，「レジスタ値をメモリにコピーする(ストア命令)こと」だけです．

CISCプロセッサでは，通常，メモリの値をレジスタの値に加えたり，レジスタの値をメモリの値に加えたりすることが可能でした．ARMはそのような「メモリからメモリへ」の処理は行いません．ARM命令はすべて，次の三つのカテゴリに当てはまります．

(1) データ処理命令

レジスタの値に対してのみ使用できます．たとえば，二つのレジスタの値を加算して結果をレジスタに格納することができます．

(2) データ移動命令

メモリ値をレジスタにコピー(ロード命令)，またはレジスタ値をメモリにコピー(ストア命令)します．システム・コードでのみ有用ですが，メモリとレジスタの値を交換する追加形式の命令があります．

(3) フロー制御命令

通常の命令実行は連続するメモリ・アドレスに格納された命令を使用します．フロー制御命令は，永久に(分岐命令)，または元のシーケンスに復帰するアドレスを保存しながら(リンク付き分岐命令)，またはシステム・コードにトラップしながら(スーパバイザ・コール)，実行を別のアドレスにスイッチさせます．

◆スーパバイザ・モード

ARMプロセッサは，保護されたスーパバイザ・モードに対応しています．保護メカニズムにより，「コードが不法な処理をしない」というチェックなしにはユーザ・コードはスーパバイザ特権を得られません．

これにより，ユーザ・レベル・プログラマは，あらかじめ決められたスーパバイザ・コールを経由してしかシステム・レベル機能にアクセスできなくなります．システム・レベル機能は，通常，ハードウェア周辺レジスタへのアクセス，文字入力および出力などよく使われる処理へのアクセスを含みます．ユーザ・レベル・プログラマは，プログラムが「所有する」データ上を操作するアルゴリズムを考案することに専念していて，プログラム外の世界との交信はOSに任せます．OS機能を必要とする命令については，3.3節の「スーパバイザ・コール」の項で述べます．

◆ARM命令セット

ARM命令は，すべて32ビット長(第7章で説明する圧縮16ビットThumb命令を除く)で，メモリ内では4バイト境界をもって並んでいます．命令セットの基本的な使用方法については第3章で，またバイナリ命令形式を含む詳細については第5章で説明します．

ARM命令セットのもっとも注目すべき特徴は次のとおりです．

(1) ロード/ストア・アーキテクチャ
(2) 3アドレス・データ処理命令(つまり，二つのソース・オペランド・レジスタと結果レジスタがすべて個別に指定される)
(3) 全命令の条件実行
(4) 非常に強力なロード/ストア複数レジスタ命令
(5) 汎用シフト演算と汎用ALU処理を1クロック・サイクルで1命令として実行する能力
(6) プログラマ・モデルに新しいレジスタとデータ型を追加できるなど，コプロセッサ命令セットを使ったオープンな命令セットの拡張
(7) Thumbアーキテクチャにおける命令セットの高密度16ビット圧縮表現

最近のRISC命令セットをご覧になられた方は，ARM命令セットがほかの市販RISCプロセッサより多くの命令の書式をもつように思われるかもしれません．たしかに，このことは，命令デコードの複雑化につながり

ますが，同時に高コード密度も可能にします．ARMプロセッサがよく使われる小さな組み込み型システムでは，このコード密度のメリットがデコード複雑化による性能上のデメリットを上回ります．Thumbはこの利点をさらに拡張し，ARMにおいてCISCプロセッサより優れたコード密度を可能にしました．

◆I/Oシステム

ARMは，**I/O**（入出力）周辺装置（ディスク・コントローラ，ネットワーク・インターフェースなど）を割り込みサポート付きでメモリ・マップされたデバイスとして扱います．これらデバイスの内蔵レジスタは，ARMのメモリ・マップ内でアドレシング可能となり，ほかのメモリと同じ（ロード/ストア）命令で読み書きできます．

周辺装置は，通常の割り込み（*IRQ*）または高速割り込み（*FIQ*）入力を使って割り込み要求をし，プロセッサの注意を引きます．割り込み入力はともにレベル・トリガで，かつマスク可能です．通常，多くの割り込みソースは*IRQ*入力を共有し，1個か2個の時間にクリティカルなソースは，優先順位の高い*FIQ*入力につなぎます．

システムの中には，広帯域幅I/Oトラフィックに対応するためにプロセッサ外付けのダイレクト・メモリ・アクセス（**DMA**）ハードウェアをもっているものもあります．これは，11.9節で詳しく述べます．

割り込みは，例外の一つの形式であり，以下のように処理します．

◆ARMの例外

ARMアーキテクチャは，割り込み，トラップ，スーパバイザ・コールまでサポートし，すべて「例外」というカテゴリの下にまとめられています．それらの一般的な処理方法はすべてに共通です．

1) PCをr14_*exc*に，CPSRをSPSR_*exc*にコピーすることによって，現在の状態を保存します（*exc*は例外タイプを表す）．

2) プロセッサ処理モードが，適切な例外モードに変わります．

3) PCが00_{16}から$1C_{16}$の間の値に設定されます．各値は例外タイプによって決まります．

変更したPCが指している位置（ベクタ・アドレス）にある命令は，例外ハンドラへの分岐を含んでいます．例外ハンドラは，`r13_exc`を使いますが，それは通常，ユーザ・レジスタを作業レジスタとして使えるように保存するため，メモリ中の専用スタックを初期化するよう初期化されています．

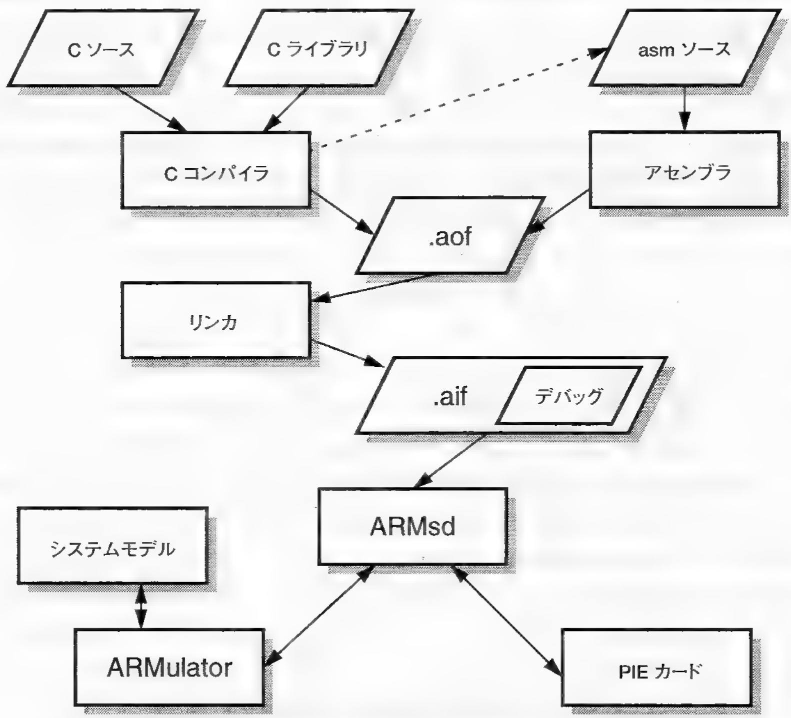
ユーザ・レジスタを復元し，PCとCPSRを自動的に復元する命令を用いれば，ユーザ・プログラムに復帰することができます．また，`r14_exc`に保存していたPC値を調整することで例外発生時のパイプライン状態を補正します．詳細は，5.2節で述べます．

2.4　ARM開発ツール

ARM用のソフトウェア開発は，ARM Limitedが一貫して開発しているツールがサポートしています．また，GNU Cコンパイラ（GCC）用のARMバックエンドなど，サードパーティ製のツールや一般的に入手可能なツールも多数あります．

ARMは，ターゲット・ハードウェアがソフトウェア開発のための環境としては適当でないような組み込みコントローラとして使われるのが一般的なので，ツールは，Windowsが搭載されるPC，適切なUNIXワークステーションなどのプラットホームで動作する**クロス開発**（つまり，コードを生成するアーキテクチャとは異なるアーキテクチャ上で動作）向けになっています．ARMクロス開発ツールキットの全体構造を図2.4に示し

ます．Cやアセンブラのソース・ファイルはARMオブジェクト形式（.aof）ファイルにコンパイルまたはアセンブルされ，ARMイメージ形式（.aif）に出力されます．イメージ形式ファイルは，ARMシンボリック・デバッガ（**ARMsd**）に必要なデバッグ・テーブルを含むことができるように作られています．ARMsdは，プログラムのロード，実行，そしてARM開発ボードなどのハードウェア上，またはARMのソフトウェア・エミュレーション（**ARMulator**）上でのデバッグが可能です．ARMulatorは，キャッシュや特殊なメモリ・タイミング特性などのシステム仕様を取り込めるようにソフトウェア・モデルの拡張が簡単にできるように設計されています．



〔図2.4〕ARMクロス開発ツールキットの構造

◆ARM Cコンパイラ

ARM CコンパイラはANSI（American National Standards Institute）の標準Cに準拠し，標準関数のライブラリを備えています．外部から使用可能な関数はすべてARMプロシージャ・コール・スタンダード（6.8節を参照）を使用しています．ARMオブジェクト形式の代わりにアセンブリ・ソース出力も可能ですから，コードを点検したり手で最適化してから順にアセンブルすることができます．

コンパイラは，Thumbコードも生成します．

◆ARMアセンブラ

ARMアセンブラはフルマクロ・アセンブラで，Cコンパイラの出力とリンクできるARMオブジェクト形式を作成します．

アセンブリ・ソース言語は機械語のレベルに近く，多くのアセンブリ命令は一つのARM（またはThumb）命令に翻訳されます．ARMアセンブリ言語プログラミングについては次章で解説し，アセンブリ言語形式を含

むARM命令セットの詳細は第5章で述べます．Thumb命令セットおよびアセンブリ言語形式については第7章にあります．

◆リンカ

リンカは，一つまたは複数のオブジェクト・ファイルを組み合わせ，実行可能なプログラムにします．リンカは，オブジェクト・ファイル間のシンボル参照を解決し，プログラムでの必要に応じてライブラリからオブジェクト・モジュールを抽出します．また，プログラムの多様なコンポーネントを，コードがRAM（Random Access Memory．読み書き可能）用なのかROM（Read Only Memory）用なのか，オーバレイが必要かどうか，などの条件に応じて様々な方法でアセンブルできます．

通常，リンカは出力ファイル中にデバッグ・テーブルを含みます．オブジェクト・ファイルが全デバッグ情報付きでコンパイルされていれば，すべてのシンボリック・デバッグ・テーブルをもつことになります（そうすれば，プログラムはソース・プログラム中のさまざまな名称を使ってデバッグできる）．リンカは，実行はできませんが，将来オブジェクト・ファイルを効率よくリンクするためのライブラリ・モジュールも作成します．

◆ARMsd

ARMシンボリック・デバッガは，エミュレータ（ARMulator），またはARM開発ボードなどターゲット・システム上でリモート動作しているプログラムのデバッグを支援するフロントエンド・インターフェースです．リモート・システムは，シリアル・ラインまたはJTAGテスト・インターフェース（8.6節を参照）を使うリモート・デバッグ・プロトコルをサポートしなければなりません．プロセッサ・コアが特定用途向けシステム・チップに組み込まれている場合は，システムのデバッグは複雑になります．これについては，第8章で説明します．

基本的には，ARMsdは実行可能なプログラムをARMulatorまたは開発ボードにロードし，実行します．また，コード内のアドレスにブレークポイントを設定できます．ブレークポイントは，プロセッサの実行を停止し，プロセッサ状態を検査できます．ARMulator，または適切にサポートされているハードウェア上で実行しているとき，ウォッチ・ポイントも設定できます．これはメモリ・アドレスで，データ・アドレスとしてアクセスさせると，同様に実行を停止します．

もっと洗練されたレベルでは，ARMsdはフルソース・レベルのデバッグに対応します．この場合は，ARMマシンやアセンブリ・コードに慣れていないCプログラマでもCソース・ファイルを使ってブレークポイントを指定でき，またオリジナル・プログラムでの変数名を使用してプログラムのデバッグができます．

◆ARMulator

ARMulator（ARM emulatorの意）は，ホスト・システム上でソフトウェア内におけるさまざまなARMプロセッサ・コアの動きをシミュレーションするプログラム・スイートです．以下のように，各精度レベルで動作可能です．

(1) 命令アキュレート・モデリングでは，プロセッサの正確なタイミング特性までは考慮しない，システム状態の動作を正確にシミュレーションする．

(2) サイクル・アキュレート・モデリングでは，サイクルごとのプロセッサの動きを正確にシミュレーションし，プログラムに必要なクロック・サイクル数が正確にわかる．

(3) タイミング・アキュレート・モデリングは，サイクル中の正しい時間に信号を表示し，回路の遅延がわかる．

上記のアプローチは，すべて実際のハードウェアより，かなりゆっくり動作しますが，最初の方法がもっとも速度の低下が少なくソフトウェア開発には最適といえます．

一番単純な方法は，ARMulatorを使ってCコンパイラまたはアセンブラを使って開発したARMプログラムをホスト・マシン上で（ARMプロセッサをつながずに）テスト，デバッグすることです．この場合，プログラムが実行するのに必要なクロック・サイクル数が正確にわかり，ターゲット・システムの性能が評価できます．

一番複雑な方法は，OSが動作していて，キャッシュやメモリ管理機能が追加されているターゲット・システムのタイミング・アキュレートなCモデルの中心としてARMulatorを使用することです．キャッシュとOSが動作しているメモリ管理機能をフルに加えたものです．

上の両極端の間にくるのが，高速プロトタイプ・メモリ・モデルとコプロセッサ・インターフェース・サポートを含む一連のARMulatorモデル・プロトタイピング・モジュールです（8.5節で詳しく述べる）．

ARMulatorは，VHDL（VHDLは標準の広く普及しているハードウェア記述言語）のような言語に基づくハードウェア・シミュレーション環境においても時間アキュレートなARM処理モデルのコアとして使えます．ARMulatorのCコードをVHDL環境にインターフェースするためには，VHDL「ラッパ」が生成されなければなりません。

◆ARM開発ボード

ARM開発ボードとは，ARM搭載システムの開発を支援するためにさまざまなコンポーネントやインターフェースを組み込んだ回路基板です．開発ボードには，ARMコア（ARM7TDMIなど），ターゲット・システムの性能とメモリのバス幅に合うよう構成可能なメモリ・コンポーネント，特定アプリケーション向けペリフェラルをエミュレートするよう構成可能な電気的にプログラマブルなデバイスが含まれます．この開発ボードは，最終的な特定アプリケーション向けハードウェアの入手前におけるハードウェア開発とソフトウェア開発の両方をサポートします．

◆ソフトウェア・ツールキット

ARM Limitedは，上に述べた完全なツール・セットをサポート・ユーティリティ・プログラムとドキュメントを付けて「ARM Software Development Toolkit」提供しています．ツールキットCD-ROMは，ほとんどのバージョンのWindows OSで動作するPCバージョンのツール・セットを含み，完全なWindowsベースのプロジェクト・マネージャも入っています．ツールキットは，ARMの新バージョンが出荷されると同時にアップデートされます．

ARMプロジェクト・マネージャは，上記ツールのためのグラフィカルなフロントエンドです．そして，特定のプロジェクトを構成するファイル群のリストから一つのライブラリもしくは実行可能なイメージを構築します．ファイルには次のようなものがあります．

・ ソース・ファイル（C，アセンブラなど）
・ オブジェクト・ファイル
・ ライブラリ・ファイル

ソース・ファイルは，プロジェクト・マネージャによって依存リストが作成され，出力ライブラリもしくは実行可能なイメージが構築されます．構築（ビルド）では，次のように多くのオプションが選択できます．

(1) 出力をコード・サイズにおいて最適化するか，実行時間で最適化するか．

(2) 出力をデバッグにするかリリース形式にするか．
（ソース・レベル・デバッグ向けにコンパイルしたコードは，完全には最適化できません．なぜなら，ソースから完全に最適化した出力のマッピングはデバッグ目的にしてはわかりにくいからです．）

(3) どのARMプロセッサがターゲットか（とくに，Thumbをサポートするかどうか）．

CD-ROMには，コマンド・ライン・インターフェースを使うSunやHP社のUNIXホストで動作するバージョンのツールも入っています．どのバージョンもオンライン・ヘルプがついています．

◆JumpStart

VLSI Technology, Inc.のJumpStartツールも同様の基本的開発ツールを含んでいますが，標準ARMツールキットのコマンドライン・インターフェースではなく，対応しているワークステーション上の完全なX-WindowインターフェースをもっていますⅠ.

ARM開発を支援するツール・サプライヤは他にも多くあります。

2.5　例題と演習

<例題2.1>　ARMアーキテクチャの主要な特徴を記述しなさい．

ARMアーキテクチャの主要な特徴は次のとおりです．

1　多数のレジスタ群があり，その全部がほぼ汎用
2　ロード/ストア・アーキテクチャ
3　3アドレス命令（つまり，二つのソース・オペランド・レジスタと結果レジスタがすべて個別に指定される）
4　全命令の条件実行
5　非常に強力なロード，ストア複数レジスタ命令
6　汎用シフト演算と汎用ALU処理を1クロック・サイクルで一つの命令として実行する能力
7　プログラマ・モデルに新しいレジスタとデータ型を加えるなど，コプロセッサ命令セットを使ったオープンな命令のセット拡張

Thumb命令セットをARMアーキテクチャの一部と考えるなら，次の特徴も加えます．

8　Thumbアーキテクチャにおける命令セットの高密度16ビット圧縮表現

演習2.1.1　ARMと他のRISCアーキテクチャの共通する特徴はどれですか？

演習2.1.2　ARMアーキテクチャの特徴のうち，他の多くのRISCアーキテクチャにないものはどれですか？

演習2.1.3　他のRISCアーキテクチャの特徴で，ARMアーキテクチャにないものはどれですか？

第3章 ARMアセンブリ言語プログラミング

ARMプロセッサは，アセンブリ・レベルでも非常に簡単にプログラムできますが，多くのアプリケーションはCやC++などの高級言語でプログラムするのが適切です．

アセンブリ言語プログラミングでは，プログラマは個々の機械語命令レベルで思考しなければなりません．ARM命令は32ビット長ですから，2^{32}つまり約40億の異なるバイナリ機械語命令をもちえます．幸い，命令空間は十分に考慮された構造をもつので，プログラマは40億のバイナリ・エンコードをすべて知る必要はありません．詳細な処理の多くをアセンブラが，プログラマに代わって行います．

本章では，ユーザ・レベルでARMアセンブリ言語プログラミングを考え，ARMのPIEカードやARMエミュレータ(たとえば，ARM開発ツールキットに入っているARMulator)上で動作する単純なプログラムの書き方を説明します．この章では基本的な命令セットを理解し，第5章でシステム・レベルでのプログラミングと，バイナリ・レベルの命令符号化などARM命令セットのさらなる詳細を述べます．

ARMプロセッサの中には，16ビット「Thumb」命令と呼ばれる圧縮された命令セットがあります．これについては，第7章で触れます．

3.1　データ処理命令

ARMデータ処理命令を使って，プログラマは算術および論理演算をレジスタのデータ値について行うことができます．ほかの命令はデータを移動するか，プログラム実行のシーケンスを制御するだけですから，データ処理命令はデータ値を変更する唯一の命令です．データ処理命令は普通，二つのオペランドを必要とし，一つの結果を出しますが，例外もあります．典型的なのは，二つの値を足して一つの結果を出す処理です．

ARMデータ処理命令に適用される規則には，次のようなものがあります．

(1) 全オペランドは32ビット長で，レジスタからくるか，もしくは命令自体の中のイミディエート値で指定される．

(2) 結果があるとすれば，32ビット長でレジスタに置く(例外：ARMの一部のバージョンにある長乗算命令は64ビットの結果を生成する．これについては5.8節で述べる)．

(3) 各オペランド・レジスタとデスティネーション・レジスタは，命令内で個々に指定される．つまり，ARMはこれらの命令に関して「3アドレス形式」を使うということである．

◆単純なレジスタ・オペランド

典型的なARMデータ処理命令は，アセンブリ言語で次のように書きます．

```
ADD  r0, r1, r2        ; r0 := r1 + r2
```

セミコロンの右側にあるのはすべてコメントであり，アセンブラからは無視されます．コメントは，コードを理解しやすくする目的でアセンブリ・ソース・コード内に記述します．

この例では，たんに二つのレジスタにある値を取り(r1とr2)，加算し，3番目のレジスタ(r0)に結果を置きます．ソース・レジスタの値は，32ビット幅で，符号なし整数または符号付き2の補数と考えます．加算により，キャリアウト，または符号付き2の補数値の場合は符号ビットへの内部オーバフローを生じることがあります．しかし，どちらの場合も無視されます．

アセンブリ言語ソース・コードを書く際は，オペランドを正しい順序で書くことに注意します．まず結果レジスタ，次に第1オペランド，最後に第2オペランドです〔ただし，可換(commutative)処理の場合は，第1と第2オペランドの順序は両方がレジスタであれば重要ではない〕．この命令を実行した場合，システム状態の唯一の変化はデスティネーション・レジスタの値(r0)だけです(後で述べるように，オプションとしてCPSRのN，Z，C，Vフラグも変化する)．

この形式で可能な他の命令を，以下に種類別にあげます．

(1) 算術演算

二つの32ビット・オペランドでバイナリ演算(加算，減算，オペランド順序を逆にした逆減算)を行います．オペランドは符号なし，または2の補数による符号付き整数です．キャリインは，CPSRレジスタ(2.3節を参照)のCビットの現在値です．

```
ADD  r0, r1, r2        ; r0 := r1 + r2
ADC  r0, r1, r2        ; r0 := r1 + r2 + C
SUB  r0, r1, r2        ; r0 := r1 - r2
SBC  r0, r1, r2        ; r0 := r1 - r2 + C - 1
RSB  r0, r1, r2        ; r0 := r2 - r1
RSC  r0, r1, r2        ; r0 := r2 - r1 + C - 1
```

ADDは単純な「加算」，ADCは「キャリ付き加算」，SUBは「減算」，SBCは「キャリ付き減算」，RSBは「逆減算」，RSCは「キャリ付き逆減算」です．

(2) ビット論理演算

入力オペランドの各ビット・ペアについて，指定されたブール論理演算を行います．最初の例では，0から31の各iの値に対してr0[i]:=r1[i]AND r2[i]となります．このとき，r0[i]はr0のi番目ビットです．

```
AND  r0, r1, r2        ; r0 := r1 and r2
ORR  r0, r1, r2        ; r0 := r1 or r2
EOR  r0, r1, r2        ; r0 := r1 xor r2
BIC  r0, r1, r2        ; r0 := r1 and not r2
```

1.2節の「ハードウェア・ゲート・レベル」で，AND，OR，XOR(ここではEOR)論理演算について触れました．最後のコードBICは「ビット・クリア」を意味し，第2オペランドの各‘1’のビットが第1オペランドの対応ビットをクリアするというものです(アセンブリ言語のコメントにおける「not」は，そのオペランドの各ビットを反転する)．

(3) レジスタ転送

第1オペランドを無視し(これはアセンブリ言語形式では省略される)，第2オペランド(場合によってはビットを反転して)をデスティネーションへ転送します．

```
        MOV   r0, r2              ; r0 := r2
        MVN   r0, r2              ; r0 := not r2
```

MVNは「否定して転送」を意味し，結果レジスタはソース・オペランドの各ビットを逆転して得られた値にセットされます．

(4) 比較演算

結果を生成せず（したがってアセンブリ言語形式では省略される），選択した演算にしたがってCPSRレジスタ（2.3節を参照）の条件コード・ビット（N, Z, C, V）のみをセットします．

```
        CMP   r1, r2              ; set  cc  on r1 - r2
        CMN   r1, r2              ; set  cc  on r1 + r2
        TST   r1, r2              ; set  cc  on r1 and r2
        TEQ   r1, r2              ; set  cc  on r1 xor r2
```

CMPは，「比較」，CMNは「否定して比較」，TSTは「(ビット)テスト」，TEQは「テスト・イコール」です．

◆イミディエート・オペランド

二つのレジスタを加算する代わりにレジスタにある定数を加算したいとします．その場合，第2ソース・オペランドをイミディエート値，つまり#に続くリテラル定数に代えます．

```
        ADD      r3, r3, #1          ; r3 := r3 + 1
        AND      r8, r7, #&ff        ; r8 := r7[7:0]
```

最初の例でもわかるように，3アドレス形式ではソースとデスティネーション・オペランドが別個に指定されていますが，それらは異なるレジスタである必要はありません．2番目の例のように，#のあとに&を付けることで，イミディエート値を16進法で表記できます．

イミディエート値は32ビット命令の内に符号化されるため，32ビット値なら何でもイミディエート値になるわけではありません．いかなる32ビット値もイミディエート値になりえるのは，1になるビットが隣接する8ビット内にあって，それを2ビットごとにシフトして得られる値です．有効なイミディエート値は次のように求められます．

$$immediate = (0 \rightarrow 255) \times 2^{2n} \quad \text{（式10）}$$
（ただし，$0 \leqq n \leqq 12$）

アセンブラは，可能ならばMOVをMVNに，ADDをSUBにそれぞれ変更し，イミディエート値を範囲内に収めるようにします．

このことは，イミディエート値に関する厳しい制約に見えますが，実際は，32ビット・ワード内の任意の4バイト位置にあるバイト値，2の任意の累乗など，もっとも一般的なケースをすべてカバーします．どんな場合でも，アセンブラは要求された値がコード化できない場合はレポートします（イミディエート値に関する制約があるのは，イミディエート値がバイナリ命令レベルで指定されているからである．詳細は第5章で説明する）．

◆シフトされたレジスタ・オペランド

データ処理を指定する3番目の方法は最初に述べた方法に似ています．ただし，第2レジスタ・オペランドは第1オペランドと組み合わせる前にシフト処理をします．例を示します．

```
        ADD   r3, r2, r1, LSL #3 ;  r3 := r2 + 8 × r1
```

これは，1クロック・サイクルで実行するARM命令であることに注意してください．多くのプロセッサは，シフト処理を別個の命令として処理しますが，ARMは1命令内に汎用ALU処理と統合します．

LSLは，「指定したビット数だけ論理左シフト」を意味し，この例では3です．0から31の任意の数が指定できますが，0はシフトそのものを無視するのと同じになります．前述と同じように，#はイミディエート値を示します．可能なシフト処理は次のとおりです．

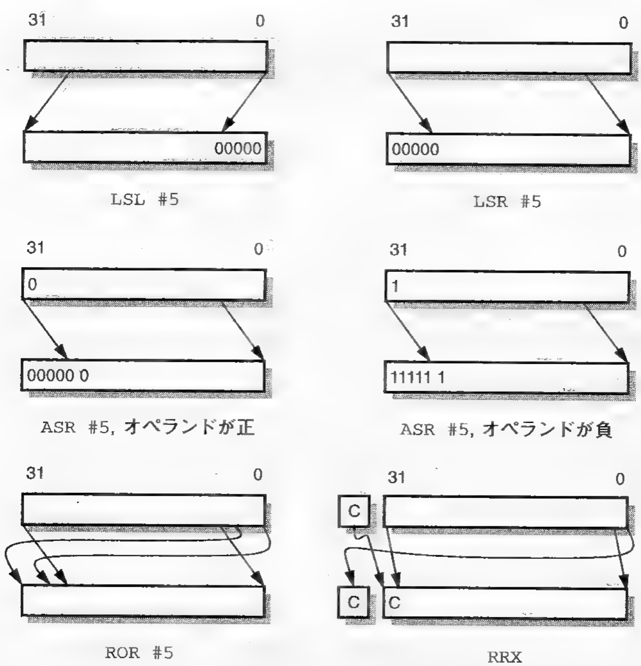
- LSL：0〜31ビットの論理左シフト．ワード最下位の空きビットは‘0’にする．
- LSR：0〜32ビットの論理右シフト．ワード最上位の空きビットは‘0’にする．
- ASL：算術左シフト．LSLと同義．
- ASR：0〜32ビットの算術右シフト．ソース・オペランドが正のときワード最上位の空きビットは‘0’にし，ソース・オペランドが負のときに‘1’にする．
- ROR：0〜32ビットの右回転．ワードの最下位から落ちたビットを使って，ワード最上位の空きビットを埋める．
- RRX：1ビット延長した右回転．空きビット（ビット31）をCフラグの古い値で埋め，オペランドを右へ1ビット・シフトする．条件コード（下記参照）を適切に使用して，オペランドとCフラグの33ビットを回転する．

上記のシフト操作を図3.1に示します．

レジスタ値を使って第2オペランドがシフトするビット数を指定することも可能です．

```
ADD   r5, r5, r3,  LSL  r2    ; r5 := r5 + r3 × 2** r2
```

これは4アドレス命令です．r2の下位の8ビットのみ有効ですが，32ビット以上のシフトはあまり使われないので，この制限はほとんど問題となりません．



〔図3.1〕 ARMのシフト操作

◆条件コードの設定

どのデータ処理命令でも，条件コード(N, Z, C, V)を設定することが可能です．比較処理は，条件コードを設定するだけなので，この選択肢はありませんが，ほかのデータ処理命令については多様な設定が可能です．アセンブリ言語レベルでは，OPコードに「条件コード設定」を意味するオプション 'S' をつけ加えます．例として次のコードは，r0-r1とr2-r3に格納した2数値の64ビット加算を実行する際，条件コード・フラグCを使って中間キャリを保存します．

```
ADDS    r2, r2, r0 ; 32-bit carry out -> C..
ADC     r3, r3, r1 ; .. and added into high word
```

OPコード拡張子Sによって，プログラマは命令が条件コードを更新するかどうかを制御できるため，必要なら条件コードを長い命令シーケンスにわたって保存できます．算術演算(ここではCMP, CMNを含む)は，演算結果にしたがってすべてのフラグを設定します．論理または移動処理はCやVにとって意味のある値を生成しませんので，結果としてNとZだけが設定されます．ただし，Vは保存し，シフト処理がないときはCを保存，シフト処理があればシフトによって端からはみ出したビットの値でCを設定します．詳細は，あまり重要ではありません．

◆条件コードの使用

Cフラグを算術データ処理命令の入力として使うことはすでに示しました．しかし，条件コードのもっとも重要な用法，つまり条件分岐命令によるプログラム・フロー制御についてはまだ述べていません．これは3.3節で説明します．

◆乗算

データ処理命令の特殊な形で，乗算をサポートします．

```
MUL     r4, r3, r2          ; r4 := (r3 x r2)[31:0]
```

以下のように，ほかの算術命令とは大きな違いがあります．

- イミディエート値の第2オペランドがサポートされていない．
- デスティネーション(結果)・レジスタが第2ソース・レジスタと同じにならない．
- Sビットが設定されている場合，Vフラグが(論理命令用として)保存され，Cフラグは意味をもたない．

二つの32ビット整数を乗算すると64ビットの結果が出ます．その最下位32ビットはデスティネーション・レジスタに置かれ，残りは無視されます．これはモジュロ2^{32}算術の乗算と考えることができ，オペランドが符号付き整数であれ符号なし整数であれ，正しい結果を出します(ARMはいくつかの長乗算命令もサポートしている．これは，最上位32ビットを第2デスティネーション・レジスタに置く．5.8節を参照)．

別の形でも制約は同じですが，結果を現在の総計に加えます．これは，積和命令です．

```
MLA     r4, r3, r2, r1      ; r4 := (r3 x r2 + r1)[31:0]
```

定数による乗算は，定数をレジスタにロードし，いずれかの命令を使って実現できます．しかし，シフト，加算，減算を用いてデータ処理命令の短い連続としたほうが通常は効率的です．r0に35を掛ける例を次にあげます．

```
ADD     r0, r0, r0, LSL #2; r0':= 5 x r0
RSB     r0, r0, r0, LSL #3; r0":= 7 x r0' (= 35 x r0)
```

3.2 データ転送命令

データ転送命令は，ARMのレジスタとメモリ間でデータを移動します．ARM命令セットには三つの基本的なデータ転送命令形式があります．

(1) 単一レジスタ・ロード/ストア命令

ARMのレジスタとメモリ間で一つのデータ項目を転送するもっとも柔軟な方法です．データ幅は，バイト，32ビット・ワード，また一部のARMチップ上では16ビットハーフ・ワードです．

(2) 複数レジスタ・ロード/ストア命令

単一レジスタ転送命令ほど柔軟ではありませんが，多量のデータを効率的に転送します．プロシージャのエントリ（入口）およびイグジット（出口），ワーク・スペース・レジスタの保存と復元，データ・ブロックのメモリ間のコピーに用いられます．

(3) 単一レジスタ・スワップ命令

1命令でロードおよびストア処理両方を効率的に行い，レジスタの値をメモリの値と交換します．ユーザ・レベル・プログラムではほとんど使われないため，このセクションでは詳しく説明しません．おもに，セマフォを実現しマルチプロセッサ・システムで共用データに対するアクセスの相互排他を確実にするため使用しますが，いま，この説明がよく理解できなくても心配はありません．

単一レジスタ・ロードおよびストア命令だけを使ってARMのどんなプログラムでも書くことは可能です．ただし，複数レジスタ転送がより効率的な場合もあるため，プログラマは理解していなくてはなりません．

◆レジスタ間接アドレシング

前述の1.4節の終わりで，プロセッサ命令セット設計者が扱えるメモリ・アドレシングのメカニズムについて触れました．ARMの転送命令はすべてレジスタ間接アドレシングに基づいており，モードにはベース・オフセット・アドレシング，ベース・インデクス・アドレシングなどがあります．

レジスタ間接アドレシングは，一つのレジスタ（ベース・レジスタ）の値をメモリ・アドレスとして用い，値をそのアドレスから別のレジスタへロードするか，別のレジスタからそのメモリ・アドレスへストアするかします．

この命令は，アセンブリ言語で次のように書けます．

```
        LDR     r0, [r1]        ; r0 := mem32[r1]
        STR     r0, [r1]        ; mem32[r1] := r0
```

アドレシングの他の形式も，ベース・アドレスにイミディエート値やレジスタ・オフセットを加えることにより，すべてこの形式の上に成り立ちます．いずれの場合も，求める転送アドレスに近いアドレスをARMレジスタにロードすることが必要です．まず，メモリ・アドレスをレジスタに入れる方法を見てみましょう．

◆アドレス・ポインタの初期化

あるメモリ位置へ，もしくはあるメモリ位置から，ロードしたりストアしたりする際，そのアドレス位置を格納するために，各ARMレジスタを初期化する必要があります．また，単一レジスタ転送命令の場合は，その位置の4Kバイト以内のアドレスを格納します（4Kバイト範囲については後で述べる）．

位置が実行しているコードの近くの場合は，プログラム・カウンタr15が目的アドレスの近くであるという

事実を利用することも可能です．小さなオフセットをr15に加算するにはデータ処理命令を使うこともできますが，適切なオフセットを計算するのはそれほど単純ではありません．しかし，アセンブラには少し巧妙な計算方法があり，ARMアセンブラは内蔵**疑似命令**ADRでこれを簡単にやってのけます．疑似(pseudo)命令は，アセンブリ・ソース・コードでは通常の命令のように見えますが，ARM命令と直接は対応していません．代わりに，アセンブリには一連の規則があり，疑似命令が使われた状況にもっとも適切なARM命令や短い命令シーケンスを選択します(ADRはつねに一つのADDまたはSUB命令にアセンブルされる)．

例として，ともにコード近くにあるTABLE 1からTABLE2へデータをコピーするプログラムを考えましょう．

```
COPY    ADR     r1, TABLE1          ; r1 points to TABLE1
        ADR     r2, TABLE2          ; r2 points to TABLE2
        ..
TABLE1                              ; < source of data >
        ..
TABLE2                              ; < destination >
        ..
```

ここでは，アセンブリ・コード内の特定のポイントに与えられた単なる名前としてラベル(COPY, TABLE1, TABLE2)を使用しました．最初のADR疑似命令によって，TABLE1の後に続くデータのアドレスをr1が格納し，同様に2番目のADR疑似命令によってTABLE2で始まるメモリのアドレスをr2が格納します．

もちろん，メモリ内データ項目のアドレスを計算するにはいろいろなARM命令が使われますが，小さなプログラム用にはADR疑似命令でも用が足ります．

◆単一レジスタのロード/ストア命令

これらの命令では，ベース・レジスタを用いて転送に使うアドレスを割り出すことができます．そのアドレスはターゲット・アドレス近くのアドレスとオフセットを含みます．オフセットは他のレジスタまたはイミディエート値でもかまいません．

先ほどは，オフセットを使わないもっとも単純な形式を見ました．

```
LDR     r0, [r1]            ; r0 := mem32[r1]
STR     r0, [r1]            ; mem32[r1] := r0
```

上記は，データ量がr1でアドレス指定される32ビット・メモリ・ワードであることを示しています．r1にあるワード・アドレスは4バイト境界上に並び，r1の二つの最下位ビットが0でなければなりません．これで，最初のワードをテーブルからテーブルへコピーできます．

```
COPY    ADR     r1, TABLE1          ; r1 points to TABLE1
        ADR     r2, TABLE2          ; r2 points to TABLE2
        LDR     r0, [r1]            ; load first value...
        STR     r0, [r2]            ; and store it in TABLE2
        ..
TABLE1                              ; < source of data >
        ..
TABLE2                              ; < destination >
        ..
```

ここで，データ処理命令を使って両方のベース・レジスタを修正し，次の転送に備えます．

```
COPY    ADR   r1, TABLE1      ; r1 points to TABLE1
        ADR   r2, TABLE2      ; r2 points to TABLE2
LOOP    LDR   r0, [r1]        ; get TABLE1 1st word
        STR   r0, [r2]        ; copy into TABLE2
        ADD   r1, r1, #4      ; step r1 on 1 word
        ADD   r2, r2, #4      ; step r2 on 1 word
        ???                   ; if more go back to LOOP
        ..
TABLE1                        ; < source of data >
        ..
TABLE2                        ; < destination >
        ..
```

両ベース・レジスタが，ワード・サイズである4(バイト)でインクリメントされていることに注意してください．ベース・レジスタがインクリメント前にワード境界で整列していれば(word-aligned)，インクリメント後でもワード境界で整列されます．

ロードおよびストア命令はすべて，この単純なレジスタ間接アドレシングだけを使うこともできます．しかし，ARM命令セットにはコードを効率よく実行するほかのアドレシング・モードもあります．

◆ベース+オフセット・アドレシング

ベース・レジスタに必ずしもアクセスすべきアドレスをもつ必要はありません．4Kバイト以下のオフセットを基底に加算(もしくは減算)して転送アドレスを割り出すことができます．

```
LDR     r0, [r1,#4]         ; r0 := mem32[r1 + 4]
```

これを**プレインデクス**(pre-indexed)アドレシング・モードといいます．これにより，一つのベース・レジスタを使って，同じメモリ領域にある多くのメモリ位置にアクセスが可能となります．

ベース・レジスタを転送アドレスをポイントするように修飾すればよいこともあります．これは，プレインデクス・アドレシングを**自動インデクス**(auto-indexing)とともに使えば可能で，プログラムは値のテーブル内を自由に移動することができます．

```
LDR     r0, [r1,#4]!        ; r0 := mem32[r1 + 4]
                            ; r1 := r1 + 4
```

感嘆符!は，命令がデータ転送を開始後，ベース・レジスタを更新すべきであることを示しています．ARMでは，自動インデクスはデータがメモリからフェッチされている間にプロセッサのデータパスで行われるので，余計な時間はまったくかかりません．自動インデクスは，単純なレジスタ間接ロードに続けて，ベース・レジスタにオフセット(この例では4バイト)を加算するデータ処理命令を行うのとまったく同じですが，余計な時間とコード・スペースはかかりません．

また，**ポストインデクス**(post-indexed)アドレシングと呼ばれる命令も使えます．これにより，オフセットなしのベース・レジスタを転送アドレスとして使用し，その後で自動インデクスできます．

```
LDR     r0, [r1], #4        ; r0 := mem32[r1]
                            ; r1 := r1 + 4
```

ここでは，感嘆符は必要ありません．なぜなら，イミディエート・オフセットはベース・レジスタ修飾子としてしか使わないからです．再度述べますが，この形式の命令は単純なレジスタ間接ロードにデータ処理命令

を続けるのとまったく同じですが，より高速でコード・スペースを小さくできます．

最後の方法を用いて，先に述べたテーブル・コピー・プログラムを改善してみます．

```
COPY    ADR     r1, TABLE1          ; r1 points to TABLE1
        ADR     r2, TABLE2          ; r2 points to TABLE2
LOOP    LDR     r0, [r1], #4        ; get TABLE1 1st word
        STR     r0, [r2], #4        ; copy into TABLE2
        ???                         ; if more go back to LOOP
        ..
TABLE1                              ; < source of data >
        ..
TABLE2                              ; < destination for >
        ..
```

ロードとストア命令は，必要な数の値がTABLE2にコピーされるまで繰り返されます．そしてループを出ます．ループの出口を決めるには，フロー制御命令が必要です．これについては，後で述べます．

上記の例で，ベース・レジスタからのアドレス・オフセットはつねにイミディエート値でした．他のレジスタでもかまいませんが，その場合はベースに加算する前にシフト処理をします．この形式は，イミディエート・オフセット形式よりも不便です．詳しくは，5.10節で説明します．

最後のバリエーションですが，転送されるデータ項目のサイズが32ビット・ワードでなく一つの符号なし8ビット・バイトであることもあります．BをOPコードに付け加えることで，このオプションが選択できます．

```
LDRB    r0, [r1]            ; r0 := mem8[r1]
```

この場合，転送アドレスは任意の境界で整列ができ，4バイト境界には限りません．なぜなら，バイトはどのバイト・アドレスにでもストア可能だからです．ロードしたバイトはr0の一番下のバイトに置き，残りのバイトは0で埋めます〔最も旧いARMプロセッサ以外は，符号付きバイトもサポートしている．バイトの最上位ビットが，値を正として扱うべきか負として扱うべきか，符号付きおよび符号なし16ビットハーフ・ワードかどうかを表示する．これらのバリアント（変形態）については，5.11節で再び命令セットの詳細を述べる際に説明する〕．

◆複数レジスタ・データ転送

大量のデータ転送の際は，いくつかのレジスタを一度に移動させるほうが効率的です．この命令を使えば16レジスタのうち任意のサブセット（もしくは全部）を1命令で転送できます．トレードオフは，使えるアドレシング・モードが単一レジスタ転送の場合より制限されることです．この種の命令の例を次にあげます．

```
LDMIA   r1, {r0,r2,r5}      ; r0 := mem32[r1]
                            ; r2 := mem32[r1 + 4]
                            ; r5 := mem32[r1 + 8]
```

転送データ項目はつねに32ビット・ワードですから，ベース・アドレス（r1）はワード境界で整列されています．

転送リストは{ }で示しましたが，r0からr15のどれでも入ることができます．リスト内の順序は重要ではなく，転送順序や命令実行後のレジスタの値に影響することはありません．しかし，リスト内ではレジスタを番号の小さい順に並べるのが一般的な習慣です．

r15（PC）がリストに入った場合は制御フローが変化するので注意してください．これに関してはフロー制

主命令の項で触れますので，ここでは詳しく説明しません．

上記の例は，これらの命令の各形式に共通な特徴を示しています．最下位レジスタは最下位アドレスに転送，もしくは最下位アドレスから転送され，ほかのレジスタはレジスタ番号順に，最初のものより上の次のワード・アドレスへ，もしくは最初のものより上の次のワード・アドレスから転送されます．しかし，最初のアドレスの求め方にはバリエーションがあり，自動インデクスも使えます(やはりベース・レジスタの後に！を加える)．

◆スタック・アドレシング

スタック・アドレシングにより，メモリ内にスタックを実装することができます．これにより，アドレシングのバリエーションが増えます．スタックとは，プログラムがコンパイルまたはアセンブルされる時点でデータ値のストアに使用するアドレスがわからない場合のメモリ割り当て，つまり単純な動的メモリ割り当てをサポートする一種のLIFO (Last-in First-out) ストアです．

例としては，引き数の値によって再帰の深度が決まる再帰関数があります．スタックは通常，線形データ構造として実装し，データが加えられるとメモリを伸ばしたり(**上昇**スタック)縮めたり(**下降**スタック)し，データが削除されると元に戻ります．**スタック・ポインタ**は，スタックにプッシュした最後の有効データ項目をポイントする(**フルスタック**)か，次のデータ項目が置かれる空スロットをポイントする(**空スタック**)かして，現在のスタック・トップのアドレスを保持します．

上の説明から，スタックには上昇，下降それぞれのフルおよび空スタックという四つのバリエーションがあることがわかります．ARMの複数レジスタ転送命令は，この4スタックすべてをサポートします．

- **フル上昇**：スタックはメモリ・アドレス増加により上昇し，ベース・レジスタは有効項目を含む最上位アドレスをポイントする．
- **空上昇**：スタックはメモリ・アドレス増加により上昇し，ベース・レジスタはスタック上の最初の空き位置をポイントする．
- **フル下降**：スタックはメモリ・アドレス減少により下降し，ベース・レジスタは有効アイテムを含む最下位アドレスをポイントする．
- **空下降**：スタックはメモリ・アドレス減少により下降し，ベース・レジスタはスタック下の最初の空き位置をポイントする．

◆ブロック・コピー・アドレシング

複数レジスタ転送命令をスタック表現で考えることもできるのですが，他の表現で考えたほうが理解しやすい場合もあります．たとえば，ブロック・コピー・アドレシングを用いてデータ・ブロックをメモリのある場所から別の場所にコピーする場合などは，アドレシングを別のメカニズムで捉えたほうがわかりやすくなります．そこで，ARMアセンブラはアドレシングのメカニズムについて二つの表現をもっています．

どちらも同じ基本的命令上にマッピングされ，相互変換的に使用できます．ブロック・コピー表現は，ベース・レジスタが保持しているアドレスの上にストアされるか下にストアされるか，また，アドレスのインクリメントまたはデクリメントが最初の値のストア前に始まるか後で始まるかに基づいています．2表現間のマッピングは，処理がロードかストアかによって決まります．詳細は**表3.1**を参照してください．

ブロック・コピー表現を**図3.2**に示します．そこでは，いくつかのバリアントがあり，メモリに3レジスタをストアする様子，自動インデクスがイネーブルされた場合にベース・レジスタを修飾する様子を示しています．命令以前のベース・レジスタ値は`r9`，自動インデクス後は`r9'`です．

命令の用法を説明するために，8ワードを`r0`がポイントする位置から`r1`がポイントする位置へコピーする二

つの命令を見てみましょう．

```
        LDMIA   r0!, {r2-r9}
        STMIA   r1,  {r2-r9}
```

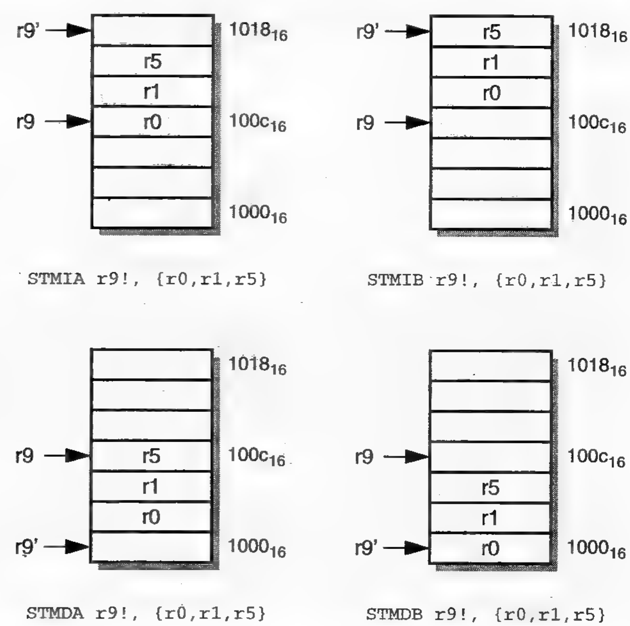
命令実行後，'！'によって8ワードを自動インデクスするため，r0は32増加しました．r1は変化しません．r2からr9が有効な値をもっている場合は，スタックにプッシュすることにより処理の間中，レジスタの値を保存することも可能です．

```
        STMEA   r13!, {r2-r9}     ; save regs onto stack
        LDMIA   r0!,  {r2-r9}
        STMIA   r1,   {r2-r9}
        LDMEA   r13!, {r2-r9}     ; restore from stack
```

ここで，最初と最後の命令に付いている接尾文字'FD'は，前にも説明したようにフル下降スタック・アドレス・モードを意味しています．スタック・ポインタがつねに一貫した動作をするよう，ほとんどのスタック処理で自動インデクスが指定されることに注意してください．

〔表3.1〕
スタックと複数ロードおよびストア命令ブロック・コピー・ビュー間のマッピング

		Ascending		Descending	
		Full	Empty	Full	Empty
Increment	Before	STMIB STMFA			LDMIB LDMED
	After		STMIA STMEA	LDMIA LDMFD	
Decrement	Before		LDMDB LDMEA	STMDB STMFD	
	After	LDMDA LDMFA			STMDA STMED



〔図3.2〕
複数レジスタ転送アドレシング・モード

ロードおよびストア複数レジスタ命令は，プロセッサ状態を保存，復元し，メモリ内でデータ・ブロックを移動する効率的な方法です．これによりコード量を節約し，単一のレジスタ・ロードまたはストア命令の同等シーケンスより4倍(命令シーケンスの動作の向上により2倍，命令数の減少により約2倍)も速く処理できます．この多大な効果を考えると，データがメモリ内でどのように構成されているかを慎重に考え，データ・アクセスするうえで複数レジスタ・データ転送命令を使用する可能性を最大にすることは十分に価値があります．

ロードおよびストア複数レジスタ命令は，別個の命令とデータ・キャッシュがあるにもかかわらず1クロック・サイクルで実行できないため，おそらく純粋なRISC命令ではありません．しかし，他のRISCアーキテクチャもプロセッサ・レジスタとメモリ間のデータ帯域幅を増やすため，複数レジスタ転送命令を採用し始めています．

ARM複数レジスタ転送命令は，非常に柔軟で，現在可視の16本のレジスタのうち任意のサブセットを転送できます．この特徴は，6.8節で説明するARMプロシージャ・コール・メカニズムで有効に利用されています．

3.3 フロー制御命令

命令カテゴリの3番目は，データの処理も移動もしません．たんに，次に実行する命令を決定します．

◆分岐命令

プログラム実行を1カ所から別の場所へスイッチするには，分岐命令を使うのが最も一般的です．

```
        B       LABEL
        ..
LABEL   ..
```

プロセッサは普通，命令を順に実行しますが，分岐命令に当たると，分岐のすぐ後の命令を実行せず，LABELの命令に直接進みます．この例では，LABELはプログラム内で分岐命令の後にありますから，間の命令はとばします．しかし，LABELは分岐の前にあることもあり，その場合，プロセッサはそこへ戻り既に実行した命令を繰り返します．

◆条件分岐

プロセッサに分岐するかどうかの決断をさせたい場合があります．たとえば，ループを実行するにはループの最初に戻る分岐が必要ですが，必要な回数のループが終わるまで，分岐が実行され，その後分岐がスキップされます．

ループ出口を制御するメカニズムを条件分岐と呼びます．この分岐には条件が付随していて，条件コードが正しい値をもつ場合にのみ分岐が実行します．典型的なループ制御シーケンスは次のとおりです．

```
        MOV     r0, #0          ; initialize counter
LOOP    ..
        ADD     r0, r0, #1      ; increment loop counter
        CMP     r0, #10         ; compare with limit
        BNE     LOOP            ; repeat if not equal
        ..                      ; else fall through
```

前の例は，条件分岐の一つ，BNE「等価でないとき分岐」を示しています．全形式は表3.2に列記しています．

表の同じ列に書いてある二つの条件（BCCとBLOの例）は，同じバイナリ・コードから発生する同義語ですが，両方あげてあるのは，特定の状況でアセンブリ・ソース・コードを解釈しやすくするためです．表に，符号付きまたは符号なしの比較とある場合は，比較命令それ自体の選択は反映せず，オペランドの別の解釈を意図します．

〔表3.2〕条件分岐

分岐	意味	用途
B BAL	条件なし つねに	いつも分岐する いつも分岐する
BEQ	等しい	比較結果が等しいかゼロ
BNE	等しくない	比較結果が等しくないかゼロでない
BPL	プラス	結果が正またはゼロ
BMI	マイナス	結果が負または否定
BCC BLO	キャリ・クリア 小さい	算術演算の結果が桁上げしない 符号なし比較で小さい
BCS BHS	キャリ・セット 大きいか等しい	算術演算の結果が桁上げする 符号なし比較で大きいか等しい
BVC	オーバフローのクリア	符号付き整数演算でオーバフローがない
BVS	オーバフローのセット	符号付き整数演算でオーバフローが起こる
BGT	大きい	符号付き整数演算でより大きい
BGE	大きいか等しい	符号付き整数演算でより大きいか等しい
BLT	小さい	符号付き整数演算でより小さい
BLE	小さいか等しい	符号付き整数演算でより小さいか等しい
BHI	大きい	符号なし比較で大きい
BLS	小さいか同じ	符号なし比較で小さいか等しい

◆条件実行

ARM命令セットのユニークな特徴ですが，条件実行は分岐のみでなく全ARM命令に適用されます．続く命令をとばすために使う分岐は，これらの命令に逆の条件を与えることで全部無視することができます．たとえば，次のシーケンスを考えてみましょう．

```
        CMP     r0, #5
        BEQ     BYPASS              ; if (r0 != 5) {
        ADD     r1, r1, r0          ;   r1 := r1 + r0 - r2
        SUB     r1, r1, r2          ; }
BYPASS  ..
```

代わりに次を使うこともできます．

```
CMP     r0, #5          ; if (r0 != 5) {
ADDNE   r1, r1, r0      ;    r1 := r1 + r0 - r2
SUBNE   r1, r1, r2      ; }
```

このシーケンスは，前出のものより小さく高速です．飛ばしたシーケンスがそれ自体の中で条件コードに何も複雑なことをしていない場合，条件シーケンスが3命令以下なら，分岐より条件実行を使うほうが効率的です．

3命令というガイドラインは，ARM分岐命令の実行が通常3サイクルかかることに基づいており，単なるガイドラインに過ぎない．コードが完全に最適化されている場合，条件実行を使うか分岐を使うかは動的なコード動作を測定して決定しなければならない．)

条件実行は，OPコード(ADDやSUBなど)の後に条件(NEやEQなど)を加えることによって起動できます(データ処理命令中で条件コードを設定するSやロードまたはストアするバイトを指定するBなどの命令修飾子の前)．

このテクニックの幅を感じさせるのは，スーパバイザ・コールやコプロセッサ命令を含むARMの全命令に条件を追加でき，条件が合わなければ飛ばすようにできることです．

条件をうまく使えば，以下のように非常に短いコードを書くこともできます．

```
; if ((a==b) && (c==d)) e++;

CMP     r0, r1
CMPEQ   r2, r3
ADDEQ   r4, r4, #1
```

最初の比較が異なるオペランドを見つけた場合，2番目が飛ばされ，インクリメントも飛ばすことに注目してください．2番目の比較を条件的にすることによって，if節中の論理積が実行されます．

◆リンク付き分岐命令

プログラムでよく求められることは，サブルーチンに分岐して，サブルーチンの完了時に元のコード・シーケンスに復帰することがあげられます．そのためには，分岐前のプログラム・カウンタの値を記録しておくことが必要です．

ARMはリンク付き分岐命令でこの機能を実現しており，分岐命令とまったく同じように分岐を実行するばかりか，リンク・レジスタr14内で分岐の次の命令アドレスも保存します．

```
        BL      SUBR            ; branch to SUBR
        ..                      ; return to here
SUBR    ..                      ; subroutine entry point
        MOV     pc, r14         ; return
```

リターン・アドレスはレジスタに保持されていますので，r14を保存せずにサブルーチンがさらに，ネストしたサブルーチンをコールしてはいけません．さもなければ，新しいリターン・アドレスが旧いものを上書きし，コール元に戻る方法がわからなくなるからです．ここで通常用いられているメカニズムは，r14をメモリのスタックにプッシュすることです．サブルーチンには普通，いくつかのワーク・レジスタが必要ですから，これらのレジスタにある旧い値は複数ストア命令を使えば同時に保存されます．

```
                BL      SUB1
                ..
SUB1            STMFD   r13!, {r0-r2,r14}    ;  save work & link regs
                BL      SUB2
                ..
SUB2            ..
```

他のサブルーチンをコールしないサブルーチン〔**リーフ**(leaf)サブルーチン〕は，r14を保存する必要はありません．r14が上書きされないからです．

◆サブルーチン復帰命令

コールするルーチンに戻るには，リンク付き分岐命令がr14に保存した値がプログラム・カウンタに再度コピーされなければなりません．プログラム・カウンタはr15として見えるので，リーフ・サブルーチンの単純なケースでは(ほかのサブルーチンをコールしないサブルーチン)MOV命令で十分です．

```
SUB2         ..
             MOV      pc, r14 ; copy r14 into r15 to return
```

実際，プログラム・カウンタがr15として見えるということは，どんなデータ処理命令もリターン・アドレスを計算するのに使えるということです．ただし，MOVがもっとも一般的に使われます．

リターン・アドレスがスタックにプッシュされた後は，複数ロード命令を用いて任意の保存されたワーク・レジスタとともに復元できます．

```
SUB1         STMFD   r13!, {r0-r2,r14}    ;  save work regs & link
             BL      SUB2
             ..
             LDMFD   r13!, {r0-r2,pc}     ;  restore work regs & return
```

リターン・アドレスが，リンク・レジスタでなくプログラム・カウンタに直接復元されることに注意してください．この一つの復元復帰命令は非常に強力です．複数レジスタ転送アドレシング・モードのスタック表現も見てみましょう．正しい値が必ず集まるように，同じスタック・モデル(この場合は，ARMコードでもっとも一般的なスタックのタイプである「フル下降」)がストアとロード両方に使われています．よほどの理由がないかぎり，どのスタックに関してもスタックを使うたびに同じアドレシング・モードを使うことが重要です．

◆スーパバイザ・コール

ディスプレイにテキストを送るなど，プログラムが入力または出力を必要とするときは，スーパバイザ・ルーチンをコールするのが普通です．スーパバイザ・ルーチンは，特権レベルで動作するプログラム，つまりユーザ・レベルのプログラムが直接できないことをするプログラムです．ユーザ・レベル・プログラムの機能限界についてはシステムによって異なりますが，多くのシステムにおいてユーザはハードウェア機能に直接アクセスできません．

スーパバイザは，特別なサブルーチン・アクセスというより，ユーザ・レベル・プログラムに見えるようにシステム・リソースにアクセスする手段を提供します．命令セットには特別な命令のSWIがあり，これらの機能をコールします(SWIはソフトウェア割り込み；Software Interruptを略したものだが，通常，「スーパバイザ・コール」と呼ばれる)．

スーパバイザ・コールはシステム・ソフトウェアに実装されており，ARMシステムごとにまったく異なる

ことがありますが，ほとんどのARMシステムは一般的なコールのサブセットと特定のアプリケーションに必要なコールを実装しています．もっともよく使われるのは，r0の下位バイトにある文字をユーザ・ディスプレイ機器に送るものです．

```
SWI      SWI_WriteC           ; output r0[7:0]
```

もう一つの便利なコールは，ユーザ・プログラムからモニタ・プログラムへ制御を戻すものです．

```
SWI      SWI_Exit             ; return to monitor
```

SWIの動作は，5.6節で詳しく説明します．

◆ジャンプ・テーブル

ジャンプ・テーブルは，通常，熟練プログラマ以外は使いませんから，アセンブリ・レベルのプログラミングを始めたばかりの読者はこの項を無視してよいでしょう．

ジャンプ・テーブルは，プログラムが計算した値に基づいて1セットのサブルーチンのうちの一つをコールしたい場合のために考えられました．これは，すでに述べた命令を使って可能です．値がr0にあるとすると，次のように書きます．

```
        BL      JUMPTAB
        ..
JUMPTAB CMP     r0, #0
        BEQ     SUB0
        CMP     r0, #1
        BEQ     SUB1
        CMP     r0, #2
        BEQ     SUB2
        ..
```

しかし，この方法では，テーブルの後ろのほうがほとんど選択されないならば話は別ですが，一般にサブルーチン・リストが長い場合は非常に時間がかかります．もっと効率的なのは，汎用レジスタ・ファイル内のプログラム・カウンタの可視性を利用することです．

```
        BL      JUMPTAB
        ..
JUMPTAB ADR     r1, SUBTAB          ; r1 -> SUBTAB
        CMP     r0, #SUBMAX         ; check for overrun..
        LDRLS   pc, [r1,r0,LSL #2]  ; .. if OK, table jump
        B       ERROR               ;  .. otherwise signal error
SUBTAB  DCD     SUB0                ; table of subroutine
        DCD     SUB1                ;       entry points
        DCD     SUB2
        ..
```

DCD命令により，アセンブラに1ワードのメモリを確保させ，命令の右の値，つまりこのケースではラベルのアドレスへ初期化します．

このアプローチでは，テーブルにどんなに多くのサブルーチンがあっても一定の性能を発揮し，それぞれの使用頻度とは関係がありません．しかし，テーブルの端を超えて読んでしまうと結果は悲惨ですから，必ずオ

ーバランをチェックしなくてはなりません．PCへのロードを条件的にすることでオーバラン・チェックをする方法を見てみましょう．こうすれば，オーバランのケースはロードをスキップしてエラー・ハンドラへの分岐に行くからです．オーバラン・チェックにかかる性能コストは最大値との比較だけです．さらに明確なコードは次のとおりです．

```
      ..
      CMP     r0, #SUBMAX       ; check for overrun..
      BHI     ERROR             ;  .. if overrun call error
      LDR     pc, [r1,r0,LSL #2] ;  .. else table jump
      ..
```

しかし，この例では，ジャンプ・テーブルを使うたびに分岐の条件的スキップの性能コストがかかります．オーバランが検知されたとき以外は，元のバージョンのほうが効率的ですが，オーバランが検知されるのはまれですし，それはエラーを表しますから，その場合の性能は重要ではありません．

ジャンプ・テーブルを使う少し特殊な用例については，6.6節の「スイッチ」の項で説明します．

3.4　簡単なアセンブリ言語プログラムを書く

簡単なアセンブリ言語プログラムを書くための基本的な道具はそろいました．どんなプログラミングでも同じですが，コンピュータに命令を打ち込む前に，アルゴリズムを明確に考えておくことが重要です．大きなプログラムでは，もちろんCやC++で書くほうがよいでしょう．ここでは，アセンブリ言語の小さなプログラム例を見ていきます．

ベテランのプログラマでも，本格的な仕事にかかる前に，簡単なプログラムが動くかどうかチェックするところから始めます．テキスト・エディタの使用を学び，アセンブラの起動法，マシンへのプログラムのロード方法，実行手順などを理解するのは大変複雑な仕事です．簡単なテスト・プログラムは，終了前にディスプレイに「Hello World」と表示するだけなので，よく「Hello Worldプログラム」と呼ばれます．ARMアセンブリ言語バージョンは次のとおりです．

```
              AREA    HelloW,CODE,READONLY ; declare code area
SWI_WriteC    EQU     &0                  ; output character in r0
SWI_Exit      EQU     &11                 ; finish program
              ENTRY                       ; code entry point
START         ADR     r1, TEXT            ; r1 -> "Hello World"
LOOP          LDRB    r0, [r1], #1        ; get the next byte
              CMP     r0, #0              ; check for text end
              SWINE   SWI_WriteC          ; if not end print ..
              BNE     LOOP                ;  .. and loop back
              SWI     SWI_Exit            ; end of execution
TEXT          =       "Hello World",&0a,&0d,0
              END                         ; end of program source
```

このプログラムは，ARMアセンブリ言語および命令セットの多くの特徴を表しています．

(1) コードAREAを適切な属性とともに明示

(2) ルーチンで使われるシステム・コールの定義（大きなプログラムでは，これは一つのファイル内に定義され，他のコード・ファイルはそれを参照する）

3. ベース・レジスタにアドレスを入れるADR疑似命令の使用
4. バイト・リストを動き回るための自動インデクス・アドレシングの使用
5. SWI命令の条件的実行で余計な分岐を防ぐ

ストリングの終わりをマークするのにゼロ・バイトを使用する（ライン・フィードとキャリッジ・リターン制御文字の後に）ことに注目してください．ループ構造を用いるときは必ず終了条件を設定してください．

このプログラムを動作するには，次のツールが必要です．すべて，ARMソフトウェア開発ツールキットに入っています．

- プログラムを打ち込む**テキスト・エディタ**
- プログラムをARMバイナリ・コードに変換する**アセンブラ**
- バイナリを実行する**ARMシステム**または**エミュレータ**．ARMシステムは，テキスト出力機能が必要です（たとえばARM開発ボードは，ホストのディスプレイに出力するためにテキスト出力をホストに送る）．

このプログラムが実行したら，もっと実用的なことをする準備は完了です．今後は，プログラム・テキストを変更していくだけです．エディタ，アセンブラ，テスト・システムまたはエミュレータの使用法は，ここまでとたいして変わりません．少なくとも，プログラムが指示を拒否し，拒否の原因がわからなくなるまではたいして変わりませんが，そうなるとデバッグを使ってプログラム内部で何が起こっているのか知る必要があります．つまり，別の複雑なツールの使い方を学ばなければならなくなりますが，ここではなるべく先に延ばすことにします．

次の例として，本書の前の部分で部分的に開発したブロック・コピー・プログラムを完成させます．うまく動いたことを確認するために文字列を使い，それをデスティネーション・アドレスから出力します．そして何か別のことにデスティネーション領域を初期化します．

```
        AREA    BlkCpy,CODE,READONLY
SWI_WriteC      EQU     &0          ; output character in r0
SWI_Exit        EQU     &11         ; finish program
        ENTRY                       ; code entry point
        ADR     r1, TABLE1          ; r1 -> TABLE1
        ADR     r2, TABLE2          ; r2 -> TABLE1
        ADR     r3, T1END           ; r3 -> T1END
LOOP1   LDR     r0, [r1], #4        ; get TABLE1 1st word
        STR     r0, [r2], #4        ; copy into TABLE2
        CMP     r1, r3              ; finished?
        BLT     LOOP1               ; if not, do more
        ADR     r1, TABLE2          ; r1 -> TABLE2
LOOP2   LDRB    r0, [r1], #1        ; get next byte
        CMP     r0, #0              ; check for text end
        SWINE   SWI_WriteC          ; if not end, print ..
        BNE     LOOP2               ;  .. and loop back
        SWI     SWI_Exit            ; finish
TABLE1  =       "This is the right string!", &0a, &0d, 0
T1END
        ALIGN                       ; ensure word alignment
TABLE2  =       "This is the wrong string!", 0
        END
```

このプログラムでは，テーブルをコピーするためにワード・ロードとストアを使用します．ですから，テーブルはワードに境界で整列されていなければなりません．その後，バイト・ロードを使い，「Hello World

プログラム」で使ったのと同じルーチンを用いて結果をプリント・アウトします.

ループ終了を制御するのに"BLT"を使うことに注目してください.TABLE1に含まれるバイト数が4の倍数でない場合,r1をぴったり一致することなくT1ENDを飛び越してしまい,"BNE"に基づいた終了条件が失敗する危険があります.

このプログラムを動作し続けることができたら,ARM命令セットの基本的操作を理解しかけているといえます.後の演習と例で理解を深めてください.さらに複雑なプログラム・タスクに挑戦していて詳細に関する疑問点が出てくることもあるでしょう.そのような疑問は,第5章で説明する完全な命令セットで解けるはずです.

◆プログラム設計

命令セットを基本的に理解したら,小さいプログラムが書け,簡単にデバッグもできます.エディタに打ち込んで様子をみるだけでいいのです.しかし,この簡単な方法だけで複雑なプログラム開発が成功すると考えるのは危険です.そのようなプログラムは,長年使用され,将来ほかのプログラムによって変更されるかもしれませんし,手に入れた顧客が予想もしなかったような方法で使うかもしれないからです.

本書はプログラム設計の教科書ではありませんが,プログラミングの基本に触れた以上,ここで指摘したいのは,「良いプログラムを書くためには,たんに座ってコードをタイプするよりもっとすることがある」ということです.

本当のプログラミングはコード化ではなく,綿密な設計で始まります.開発プロセスの第1歩は必要条件を理解することです.プログラマが必要条件を十分理解していなかったために,プログラムが思ったように動かないことは驚くほど多いのです.その場合は,必要条件(インフォーマルなことが多い)を,明確な仕様に翻訳します.そこで初めて設計を開始し,プログラム構造,プログラムが扱うデータ構造,データに必要な処理をするアルゴリズムを定義できます.アルゴリズムは,**疑似コード**(pseudo-code),つまりプログラミング言語の文法にはしたがっていませんが,意味を明確にするプログラムに似た表記法を使って表現されます.

設計が進んだ時点で,初めてコーディングが始められます.それぞれのモジュールをコーディングし,完全にテストして(「テスト・ハーネス」として設計された特別プログラムを必要とする場合もある)ドキュメント化し,プログラムを部分ごとに組み立てていきます.

現在,ほとんどのプログラミングは高級言語に基づいていますから,ここで説明するようなアセンブリ・プログラミングを使って大きなプログラムが作られることはまれです.しかし,クリティカルなアプリケーションに最高の性能を発揮させるためにアセンブリ言語で小さなソフトウェア・コンポーネントを開発することが必要な場合もあります.ですから,このような場合に備えてアセンブリ・コードの書き方を知っておくと便利です.

3.5 例題と演習

命令セットの基本を理解した後,プログラムの書き方を学ぶ近道は例を見ることです.そして,それとは少し違うことをするプログラムを自分で書いてみましょう.書いたプログラムが動くかどうか見るためには,ARMアセンブラと,ARMエミュレータかARMプロセッサが入ったハードウェアが必要です.ここでは,ARMプログラムの例とその修正案に触れます.まず,元のプログラムを動作させ,その後,演習で示された変更された機能を実行するようにエディットできるかどうか試してみましょう.

<例題3.1> r1を16進法でプリントアウトしなさい.

これは，レジスタを16進法（基数16）でディスプレイに表示する便利で簡単なルーチンです．レジスタ値を書き出し，アルゴリズムが期待する結果を生成しているかチェックすることによって，プログラムをデバッグするのに使います．もっとも，多くの場合，プログラム内部で起こっていることを知るにはデバッガを使うのが良い方法です．

```
            AREA    Hex_Out,CODE,READONLY
SWI_WriteC          EQU     &0          ; output character in r0
SWI_Exit            EQU     &11         ; finish program
            ENTRY                       ; code entry point
            LDR     r1, VALUE           ; get value to print
            BL      HexOut              ; call hexadecimal output
                    SWI_Exit            ; finish
VALUE       DCD     &12345678           ; test value
HexOut      MOV     r2, #8              ; nibble count = 8
LOOP        MOV     r0, r1, LSR #28     ; get top nibble
            CMP     r0, #9              ; 0-9 or A-F?
            ADDGT   r0, r0, #"A"-10     ; ASCII alphabetic
            ADDLE   r0, r0, #"0"        ; ASCII numeric
            SWI     SWI_WriteC          ; print character
            MOV     r1, r1, LSL #4      ; shift left one nibble
            SUBS    r2, r2, #1          ; decrement nibble count
            BNE     LOOP                ; if more do next nibble
            MOV     pc, r14             ; return
            END
```

演習3.1.1 上のプログラムを変更して，r1をバイナリ形式で出力しなさい．例のプログラムでr1にロードした値に対しては，次が得られるはずです．

```
00010010001101000101011001111000
```

演習3.1.2 HEXOUTをプログラムのベースに使って，メモリ領域の内容を表示しなさい．

<例題3.2> コール直後にある文字列を出力するサブルーチンを書きなさい.

テキスト用の別のデータ領域を作らずに，文字列を出力できると便利です（ただし，StrongARMのようにプロセッサにデータ・キャッシュと命令キャッシュが別個にある場合は，非効率です．その場合は，別のデータ領域を作ります）．コールは次のようになります．

```
BL      TextOut
=       "Test string",&0a,&0d,0
ALIGN
..                          ; return to here
```

注意すべきなのは，サブルーチンからの復帰は，コールによってリンク・レジスタに入った値へ直接行ってはいけないことです．そうすると，プログラムは文字列に入ってしまいます．適切なサブルーチンとテスト・ハーネスは次のとおりです．

```
            AREA     Text_Out,CODE,READONLY
SWI_WriteC           EQU      &0          ; output character in r0
SWI_Exit             EQU      &11         ; finish program
            ENTRY                         ; code entry point
            BL       TextOut              ; print following string
            =        "Test string",&0a,&0d,0
            ALIGN
            SWI      SWI_Exit             ; finish
TextOut     LDRB     r0, [r14], #1        ; get next character
            CMP      r0, #0               ; test for end mark
            SWINE    SWI_WriteC           ; if not end, print..
            BNE      TextOut              ;  .. and loop
            ADD      r14, r14, #3         ; pass next word boundary
            BIC      r14, r14, #3         ; round back to boundary
            MOV      pc, r14              ; return
            END
```

この例では，r14が文字列に沿ってインクリメントし，復帰前に次のワード境界に調整される様子を示します．調整（3を加え，下位の2ビットをクリアする）の手間が少しで済みそうであればチェックしてください．たった4ケースしかありません．

演習3.2.1 この例と前の例のコードを使って，以下のような形式でARMレジスタを16進数でダンプするプログラムを書きなさい．

```
r0 = 12345678
r1 = 9ABCDEF0
```

演習3.2.2 手を加えなければならないレジスタを変更前に保存しなさい．たとえば，PC相対アドレシングを使って，コード近くに保存しなさい．

第4章 ARMの構成と実装

ARM整数プロセッサ・コアは，1983-1985年にAcorn Computersで最初に開発された3μmデバイスから，1990-1995年にARM Limitedで開発されたARM6，ARM7まで，ほとんどその構成が変化していません．3段パイプラインは着実に強化され，この間にCMOSプロセス技術はダイ・サイズを大幅に小さくしました．コア性能も飛躍的に伸びましたが，処理の基本原則はほぼ同じです．

1995年以来，3段および5段パイプライン，独立した命令メモリとデータ・メモリ(通常，命令とデータで共通に使うメイン・メモリ・システムに接続された別々のキャッシュという形をとる)を使って大幅に性能を向上した新しいARMコアがいくつか発表されました．

本章では，これら2種類の基本型プロセッサ・コアの内部構造と，3段および5段パイプライン動作の一般原則を説明します．また，それら多数の実装の詳細についても触れます．特定のコアについての詳細は，第9章で述べます．

4.1 3段パイプラインARMの構成

3段パイプラインをもつARMの構成を図4.1に示します．おもなコンポーネントは次のとおりです．

(1) プロセッサ状態を格納する**レジスタ・バンク**

任意のレジスタにアクセスする二つの読み込みポートと一つの書き込みポート，プログラム・カウンタr15に特別にアクセスする追加読み込みポートと追加書き込みポート一つずつがあります(r15用の追加書き込みポートにより命令フェッチ・アドレスがインクリメントしたときにr15が更新し，追加読み込みポートにより命令フェッチがデータ・アドレス発行後に再開する)．

(2) 任意のビット数だけオペランドをシフトまたは回転する**バレル・シフタ**

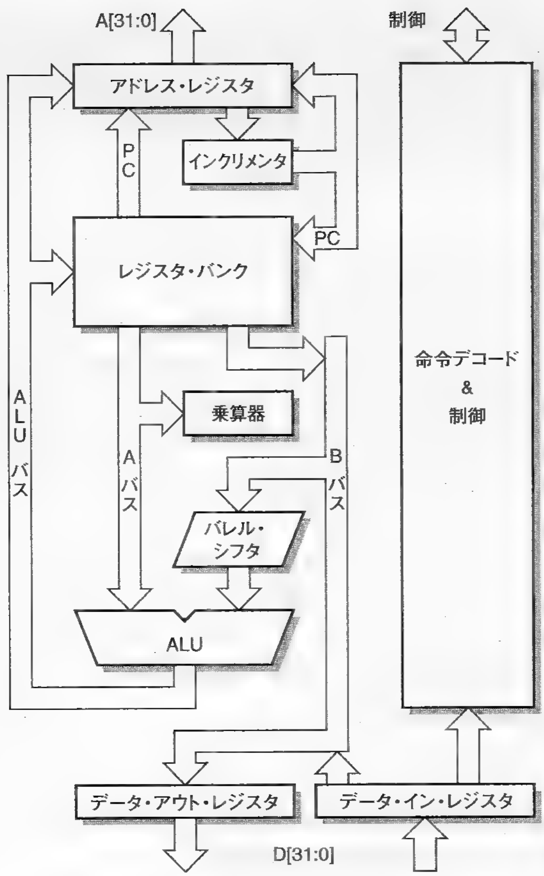
(3) 命令セットが要求する算術および論理関数を行う**ALU**

(4) 全メモリ・アドレスを選択および保持し，必要があればシーケンシャル・アドレスを生成する**アドレス・レジスタ**と**インクリメンタ**

(5) メモリへ渡す，もしくはメモリからもらうデータを保持する**データ・レジスタ**

(6) **命令デコーダ**と**関連する制御論理**

単一サイクル・データ処理命令では，二つのレジスタ・オペランドにアクセスします．Bバス上の値はシフトし，ALUでAバス上の値と組み合わされます．結果は，レジスタ・バンクに再度書き込まれます．プログラム・カウンタ値はアドレス・レジスタにあり，そこからインクリメンタに入れ，インクリメントした値はレジスタ・バンクのr15にコピーして返し，また次の命令フェッチ用のアドレスとして使うためにアドレス・レジスタにもコピーされます．



〔図4.1〕3段パイプラインをもつARM構成

◆3段パイプライン

ARM7までのARMプロセッサは，シンプルな3段パイプラインを採用しており，次のパイプライン・ステージをもちます．

(1)フェッチ

命令をメモリからフェッチし(取り込み)，命令パイプラインに置きます．

(2)デコード

命令をデコード(解読)し，データパス制御信号を次のサイクルに向けて準備します．

(3)実行

命令がデータパスを「占有」します．レジスタ・バンクを読み，オペランドをシフトし，ALU結果を生成してデスティネーション・レジスタに再度書き込みます．

いかなるときでも，三つの異なる命令が各ステージを占めるので，各ステージのハードウェアは独立した処理を行う能力がなければなりません．

プロセッサが単純なデータ処理命令を実行するとき，パイプラインによって1クロック・サイクルごとに1命令が完了します．個々の命令は完了に3クロック・サイクルかかりますから，3サイクルの**遅延時間**(latency)がありますが，**スループット**は1サイクルごとに1命令です．1サイクル命令のパイプライン処理は，図4.2に

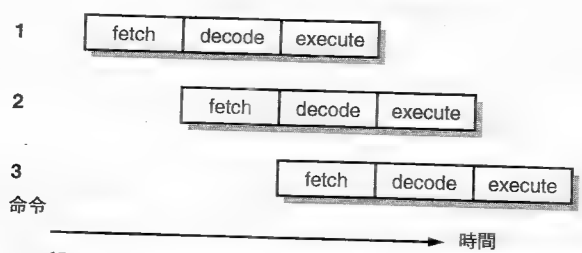
示してあります．

マルチ・サイクル命令を実行すると，フローは図4.3のように多少不規則になります．この図では，最初のADDの後にデータ・ストア命令STRが続く1サイクル命令ADDのシーケンスを示しています．主メモリにアクセスするサイクルには薄い影がつけてあり，メモリが毎サイクル使われていることがわかります．データパスも，全実行サイクル，アドレス計算，データ転送に関連して毎サイクル使われています．デコード論理はつねに，データパスが次サイクルで使う制御信号を生成します．したがって，明示的デコード・サイクルに加えて，STRのアドレス計算サイクルの間のデータ転送用の制御も生成します．

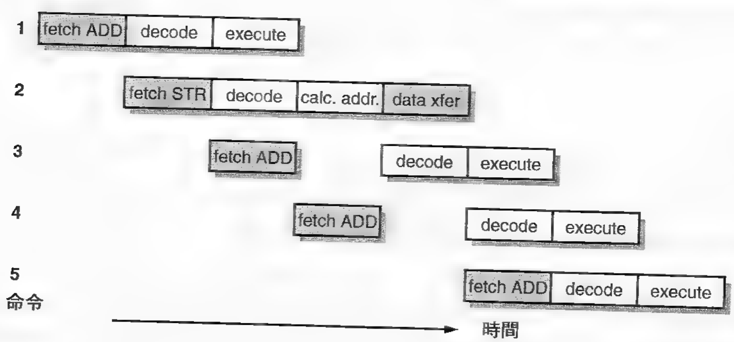
このように，この命令シーケンスでは，プロセッサの全部分が毎サイクルでアクティブになっており，逆にいえばメモリはシーケンスに必要なサイクル数を定義する制限要素になっていることになります．

以下に，ARMパイプラインで空きが入るのを見分ける簡単な方法をまとめました．

- 全命令が1以上の隣接サイクルの間，データパスを占有している．
- 命令がデータパスを占有している各サイクルで，直前サイクルのデコード論理を占有している．
- 最初のデータパス・サイクルの間，各命令が次の次の命令用のフェッチを発する．
- 分岐命令が命令パイプラインをフラッシュし，リフィルする．



〔図4.2〕ARM1サイクル命令のパイプライン処理



〔図4.3〕ARMマルチ・サイクル命令のパイプライン処理

◆PCの動作

ARMで用いられているパイプライン実行モデルの結果として，ユーザから可視であるプログラム・カウンタr15は現在の命令の先を走らなければなりません．上記のように，第1サイクルの間に命令が次の次の命令をフェッチする場合，PCは現在の命令の8バイト（2命令）先をポイントしなければならないことになります．

これは実際に起こることで，直接PC（r15）にアクセスしようとするプログラマは，ここでパイプラインの動作を考慮に入れておかなければなりません．しかし，たいていの場合は，アセンブラかコンパイラに処理をまかすことができます．

r15が命令の第1サイクルより後に使われる場合，さらに複雑な操作が必要です．なぜなら，命令自体が第1サイクルの間にPCをインクリメントするからです．そのようなPCの用法はあまり有益ではなく，ARMアーキテクチャ定義はその結果を「予想できない」と指定しています．とくに最近のARMはこれらの状況でまた違った動作をしますから，避けなければなりません．

4.2 5段パイプラインARMの構成

あらゆるプロセッサは，より高い性能に対する需要に応えるために進歩しなければなりません．ARM7までのARMで使用している3段パイプラインは非常に効果的ですが，もっと高い性能を出すためにはプロセッサ構成を見直す必要があります．与えられたプログラムの実行時間T_{prog}は，以下の式で求められます．

$$T_{prog} = \frac{N_{inst} \times CPI}{f_{clk}} \qquad \text{(式11)}$$

この式で，N_{inst}はプログラム中の実行されるARM命令数，CPIは1命令当たりの平均クロック・サイクル数，F_{clk}はプロセッサのクロック周波数です．与えられたプログラムについてN_{inst}は定数ですから(与えられたコンパイラを使い，与えられた一連の最適化を使用するなどしてコンパイルされている)，性能を向上するには二つの方法しかありません．

(1) クロック速度であるf_{clk}を増加する．

これには，各パイプライン・ステージにおける論理を単純化し，パイプライン段数を増加する必要があります．

(2) 1命令当たりの平均クロック・サイクル数であるCPIを低減する．

これには，3段パイプラインARMで複数のパイプライン・スロットを使用する命令を再実装して使用するスロット数を減らすか，命令間の依存性によって起こるパイプラインのストールを低減するか，その両方を組み合わせるかします．

◆メモリ・ボトルネック

3段コアに対してCPIを低減するうえで根本的な問題は，フォン・ノイマン・ボトルネックに関連しています．つまり，命令とデータを一つのメモリをもつプログラム内蔵方式コンピュータでは必ず，使用できるメモリ帯域幅によって性能が制限されるのです．3段ARMコアは，(ほとんど)毎クロック・サイクルで命令をフェッチしたりデータを転送したりするためメモリにアクセスします．メモリを使用していない数サイクルを削減したところで，わずかな性能向上しか得られません．CPIを大幅に向上するには，一つのメモリから1サイクル当たり32ビット以上を出すか，命令アクセスとデータ・アクセスで別々のメモリを使用するかによって，メモリ・システムが各クロック・サイクルで二つ以上の値を出す必要があります．

このような理由で，高性能ARMコアは5段パイプラインを採用し，命令メモリとデータ・メモリを別にもっているのです．命令実行を3分割ではなく5分割すれば，1クロック・サイクルで完了しなければならない最大処理量が減少するため，高いクロック周波数を使用することができます(ただし，他のシステム・コンポーネント，とくに命令メモリもこの高いクロック速度で動作するよう再設計するものとする)．また，別々の命令メモリとデータ・メモリ(共通の命令兼データ・メイン・メモリ・システムに接続した別々のキャッシュであることもある)により，コアのCPIが大幅に低減できます．

一般的な5段ARMパイプラインとは，ARM9TDMIに採用されているものです．ARM9TDMIの構成は，図

4.4に示したとおりです.

◆5段パイプライン

5段パイプラインを使用するARMプロセッサには，以下のパイプライン・ステージがあります.

(1) フェッチ

命令をメモリからフェッチし(取り込み)，命令パイプラインに置きます.

(2) デコード

命令をデコード(解読)し，レジスタ・ファイルからレジスタ・オペランドを読み出します．レジスタ・ファイルには三つのオペランド読み出しポートがあるため，ほとんどのARM命令は全オペランドを1サイクルで調達することができます.

(3) 実行

オペランドをシフトし，ALU結果を生成します．命令がロードかストアの場合，ALUがメモリ・アドレスを計算します.

(4) バッファ/データ

必要に応じてデータ・メモリにアクセスします．その必要がなければ，たんにALU結果を1クロックの間バッファし，全命令について同じパイプライン・フローにします.

(5) ライトバック

メモリからロードした全データを含め，命令が生成した結果をレジスタ・ファイルにライトバックします.

5段パイプラインはこれまで多くのRISCプロセッサに使用され，そのようなプロセッサを設計する「クラシック」な方法と考えられています．ARM命令セットは5段パイプラインを念頭に置いて設計されたわけではありませんが，比較的簡単に対応します．図4.4に示した構成におけるARM命令セット・アーキテクチャへの基本譲歩は，レジスタ・ファイルの三つのソース・オペランド読み出しポートと二つの書き込みポート(「クラシック」なRISCには二つの読み出しポートと一つの書き込みポートがある)のほか，アドレスをインクリメントするハードウェアを実行ステージに組み込むことによって複数ロードおよび複数ストア命令をサポートしていることです.

◆データ・フォワード

5段パイプラインが(3段パイプラインに比べて)複雑であるおもな原因は，命令実行が三つのパイプライン・ステージに分散していることです．パイプラインをストールすることなくデータの依存性を解決するには，フォワーディング・パスを採用するしかありません.

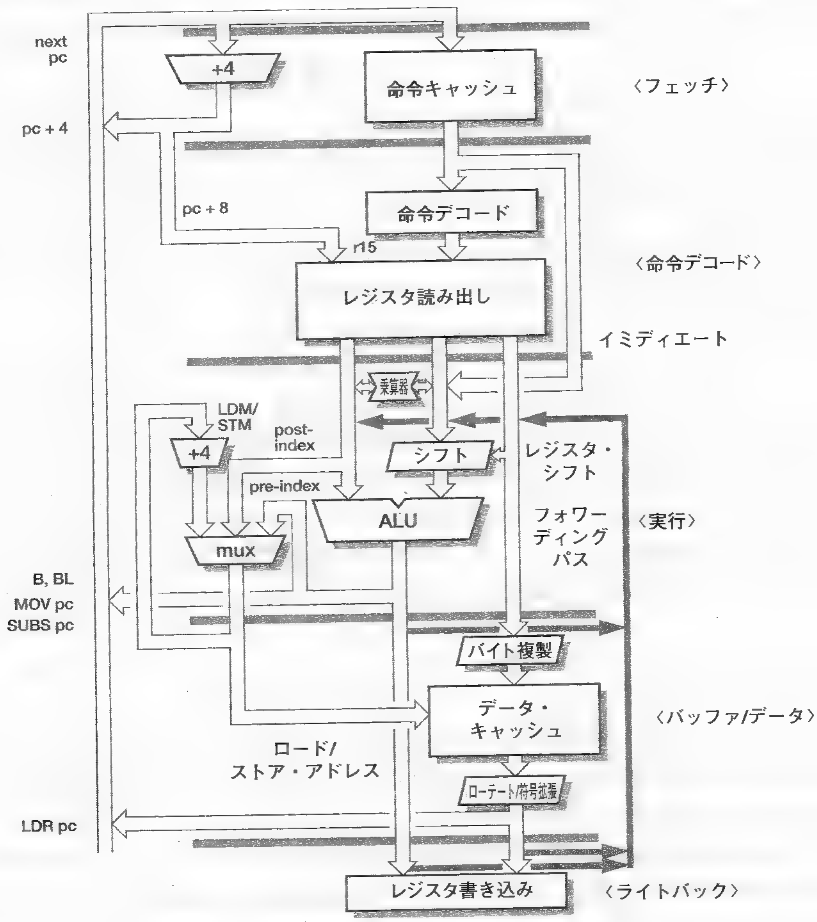
ある命令において，先行する命令のうち一つの結果を使用する必要があり，しかもその結果がレジスタ・ファイルに戻る前に使用しなければならない場合，データの依存性が発生します．(この問題については，第1章「パイプライン障害」の項で述べた)フォワーディング・パスを使えば，結果が出ると同時にステージ間を受け渡すことができます．そして5段ARMでは，三つのソース・オペランドのそれぞれを図4.4に示した中間結果レジスタのいずれかからフォワードすることが必要です.

フォワードを用いてもパイプライン・ストールを避けられない場合も一つあります．以下のコード・シーケンスを考えてください.

```
LDR  rN, [..]        ; load rN from somewhere
ADD  r2, r1, rN      ; and use it immediately
```

rNにロードした値はバッファ/データ・ステージの終わりでしかプロセッサに入らないのに，次の命令は実行ステージの最初でその値を必要としますから，プロセッサは1サイクルの間，ストールせざるをえません．このストールを防ぐ方法はただ一つ，コンパイラ（またはアセンブリ言語プログラマ）が依存性のある命令をロード命令の直後に置かないよう心がけることです．

このコード・シーケンスは，3段パイプラインARMコアに対しては悪影響がありませんから，既存のARMプログラムではよく使用されます．そのようなプログラムは5段ARMコアでも正常に動作しますが，依存性を避けるよう命令の順番を変えて書き直せば，恐らくずっと速く動作するようになります．



〔図4.4〕ARM9TDMI 5段パイプライン構成

◆PC生成

プログラマから見たr15の動作は，「PCの動作」の項でも説明しましたが，3段ARMパイプラインの動作特性に基づいています．5段パイプラインでは，命令オペランドをパイプラインにおいて1ステージ早く読み出すため，当然ながら異なる値が得られます（PC+8ではなくPC+4）．しかし，これでコードの互換性が崩れてしまうのは許されませんから，5段パイプラインARMはどれも旧い3段設計の動作を「エミュレート」します．図4.4を見ると，インクリメントしたPC値はフェッチ・ステージから直接デコード・ステージのレジスタ・ファ

イルにフィードされ，二つのステージ間のパイプライン・レジスタは飛ばします．次の命令のPC+4は，進行中の命令のPC+8に等しいため，ハードウェアを追加しなくても正しいr15の値が得られることになります．

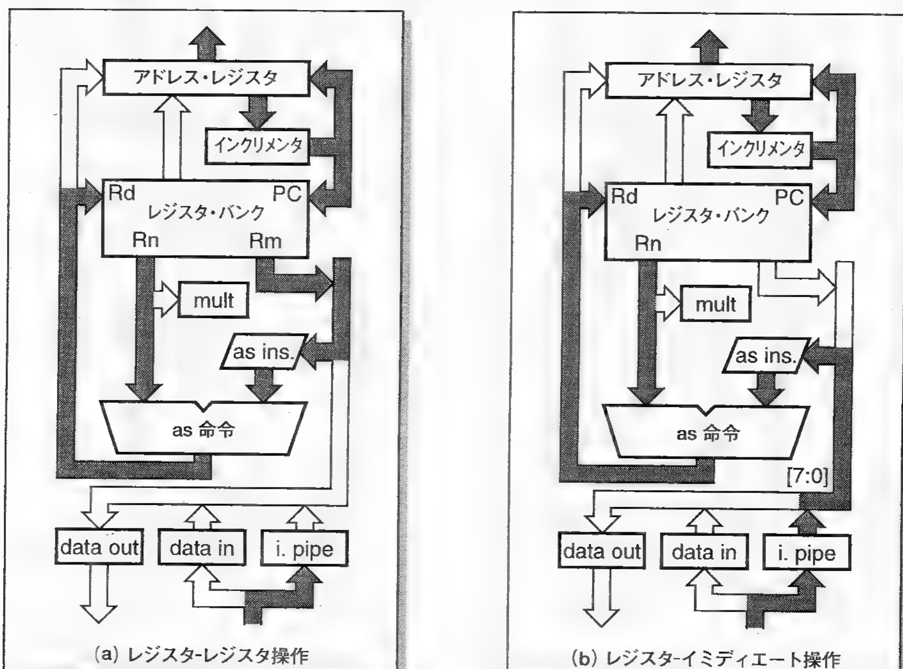
4.3 ARM命令の実行

ARM命令の実行は，図4.1に示したデータパス構成を参照すると一番よく理解できます．ここでは，注釈付きの表を使い，制御論理の部分はとばして，アクティブなバスにハイライトをつけてオペランドがプロセッサの各ユニット間で動く様子を見ましょう．まず，簡単なデータ処理命令から始めます．

◆データ処理命令

データ処理命令には二つのオペランドが必要です．片方はつねにレジスタ，もう片方は2番目のレジスタまたはイミディエート値です．2番目のオペランドはバレル・シフタを通り汎用シフト操作をします．その後，ALUで汎用ALU操作を使って最初のオペランドと組み合わせます．最後に，ALUからの結果をデスティネーション・レジスタに再度書き込みます(条件コード・レジスタが更新される)．

これらすべての処理は，図4.4にあるように，1クロック・サイクルで実行します．注目すべきは，アドレス・レジスタのPC値がインクリメントし，アドレス・レジスタとレジスタ・バンク内r15の両方に再度コピーすることです．また，次の次にあたる命令は命令パイプライン(*i.pipe*)の下部にロードします．必要なら，命令パイプラインのトップにある現在の命令からイミディエート値を取り出します．データ処理命令では，命令の下位8ビット(ビット[7:0])だけをイミディエート値に使用します．



〔図4.5〕データ処理命令のデータパスの流れ

◆データ転送命令

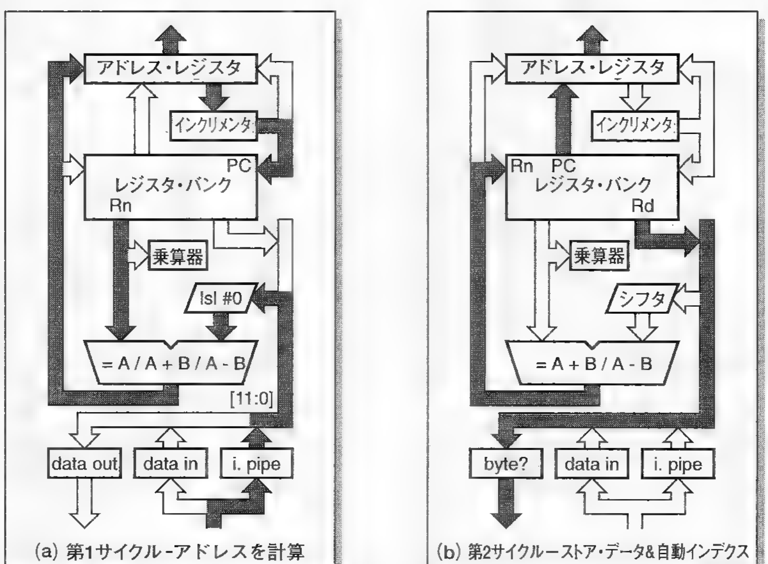
データ転送（ロードまたはストア）命令は，データ処理命令が結果を計算するのと非常に似た用法で，メモリ・アドレスを計算します．レジスタがベース・アドレスとなり，ベース・アドレスに別のレジスタかイミディエート値であるオフセットを加算（またはベース・アドレスから減算）します．しかし今回は，シフトされた8ビット値ではなく，シフト処理なしの12ビットのイミディエート値を用います．アドレスはアドレス・レジスタに送り，第2サイクルで転送します．データ転送サイクルの間，データパスをほうっておくのではなく，ALUが第1サイクルからのアドレス・コンポーネントを保持し，必要なら自動インデクスによるベース・レジスタ修飾を計算します（自動インデクスが必要なければ，計算した値は第2サイクルでベース・レジスタに書き込まれない）．

イミディエート・オフセットを使ったデータ・ストア命令（STR）における2サイクル間のデータパス処理は，図4.6に示しました．第1サイクルの最後でインクリメントしたPC値をレジスタ・バンクにストアし，第2サイクルのデータ転送アドレスを受け入れられるようアドレス・レジスタを空ける様子に注目してください．第2サイクルの終わりでは，PCがアドレス・レジスタに再び入り，命令プリフェッチを続けます．

あるサイクル中にアドレス・レジスタに送る値は，次のサイクルでメモリ・アクセスに使う値だということを，この段階で心に留めておきましょう．アドレス・レジスタは，事実上，プロセッサ・データパスと外部メモリとの間のパイプライン・レジスタなのです（どうしても必要ならば，アドレス・レジスタはパイプライン遅延の責任をメモリにまわし，現在のサイクルの終わり直前で次のサイクルのメモリ・アドレスを生成できる．これにより，メモリ・デバイスの中には高性能を出すものもあるが，詳細は後で触れる．当面は，アドレス・レジスタをメモリへのパイプライン・レジスタとして考える）．

命令がバイト・データ・タイプのストアを指定している場合，“data out”ブロックがレジスタから最下位バイトを取り出し，32ビット・データ・バス上に4回にわたって複製します．そして，外部メモリ制御論理がアドレスバスの最下位の2ビットを使ってメモリ・システム内で適切なバイトをアクティブにします．

ロード命令も同様のパターンをたどりますが，メモリからのデータは第2サイクルで“data in”レジスタまでしか行かず，第3サイクルでそこからデスティネーション・レジスタまでデータを転送しなければなりません．



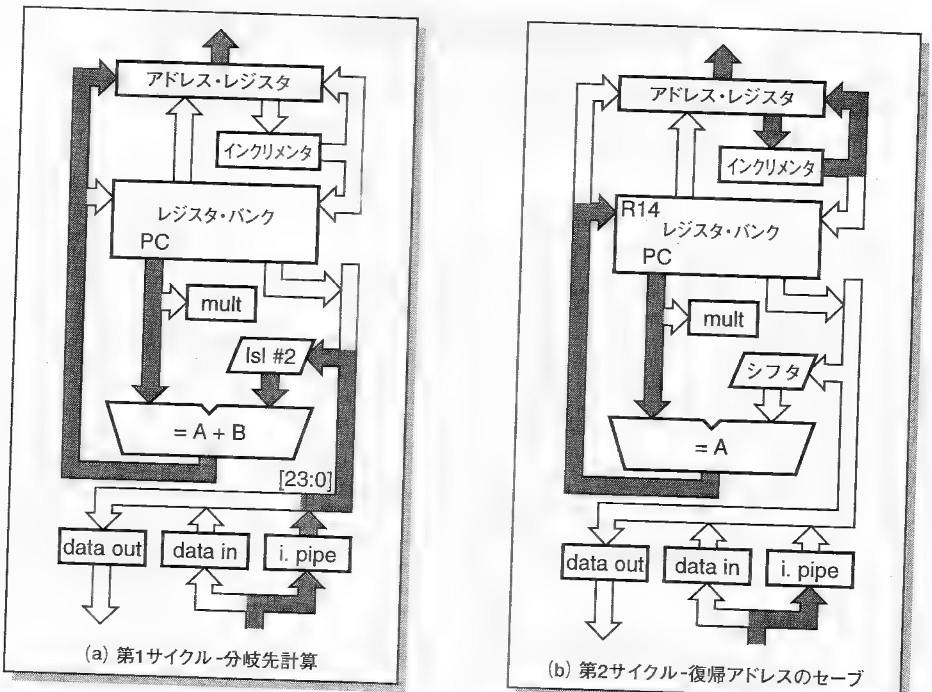
〔図4.6〕STR（ストア・レジスタ）のデータパスの流れ

◆分岐命令

分岐命令は，図4.7に示すように，第1サイクルでターゲット・アドレスを計算します．命令から24ビットのイミディエート・フィールドが取り出され，左へ2ビット・シフトしてPCに加えるワード境界のオフセットを出します．結果は命令フェッチ・アドレスとして発行し，必要であれば(つまり，命令が「リンク付き分岐」であれば)，命令パイプラインがリフィルしている間に復帰アドレスをリンク・レジスタ(r14)にコピーします．

第3サイクルはパイプライン・リフィルを完了するのに必要ですが，同時に，リンク・レジスタにストアされた値を少し訂正して，分岐に続く命令を直接ポイントさせるのにも使用します．これは，次の命令のアドレスはpc+4であるのに，r15がpc+8を含んでいるため(4.1節の「PCの動作」の項を参照)必要なのです．

他のARM命令も，上記と同様に働きます．次に，データパスがどのようにこれらの処理を実行するのか，さらに詳細を見てみましょう．



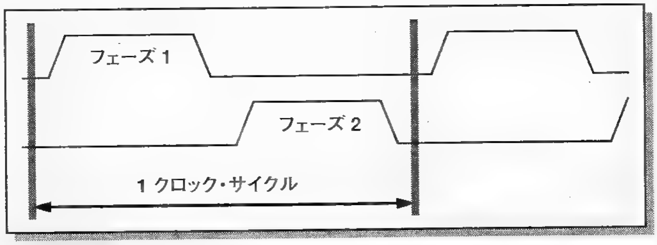
〔図4.7〕 分岐命令の(3サイクルのうち)最初の2サイクル

4.4 ARMの実装

ARMの実装は，第1章でMU0について説明したのと同様のアプローチをとります．設計は，レジスタ転送レベル(RTL)表記で記述されるデータパス・セクションとステートマシン(FSM)として捉らえられる制御セクションに分かれます．

◆クロッキング方式

1.3節で紹介したMU0の例とは違って，ARMはエッジ・トリガのレジスタをもっていません．その代わり，設計のベースとなるのは，図4.8のように，単一入力クロック信号から内部で生成する2相の非オーバラップ・クロックです．この方式により，レベル・トリガのトランスペアレント・ラッチの使用が可能です．データ移動は，フェーズ1でオープンであるラッチとフェーズ2でオープンであるラッチの間を交互に行き来することによって制御されます．クロックのフェーズ1とフェーズ2の重ならない性質のおかげで，回路の競合状態がありません．



〔図4.8〕2相非オーバラップ・クロック方式

◆データパス・タイミング

データパス・コンポーネントの通常のタイミングを図4.9に示します．レジスタ読み込みバスはダイナミック動作で，フェーズ2の間にプリチャージされます(ここでの「ダイナミック」とは，場合によりバスが非駆動で論理値を電荷として保持することを指します．電荷保持回路により疑似スタティック動作をし，クロックがサイクルのどの時点で止まってもデータは失われません)．フェーズ1が“H”になると，選択したレジスタがフェーズ1の早期に有効になる読み込みバスを放電します．一つのオペランドがバレル・シフタを通過しますが，ここでもダイナミック動作テクニックを用います．そして，シフタ出力はフェーズ1の少し後で有効になります．

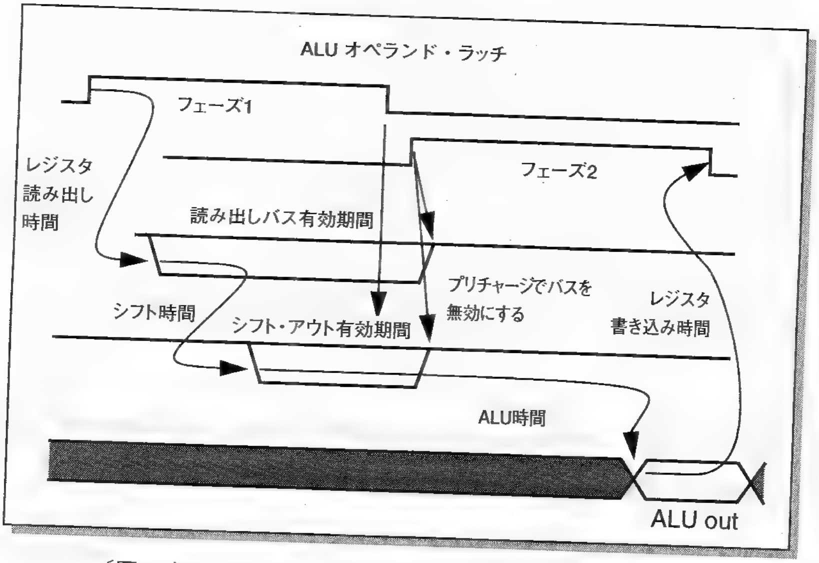
ALUにはフェーズ1でオープンになる入力ラッチがあるため，オペランドは有効になると同時にALU内で組み合わされます．しかし，ラッチはフェーズ1の終わりでクローズするため，フェーズ2のプリチャージはALUに届きません．そして，ALUはフェーズ2の間もオペランド処理を続けます．ALUはフェーズの終わりまで有効な値を生成し，その値はフェーズ2の終わりにデスティネーション・レジスタにラッチされます．

データはALU入力ラッチを通過しますが，データ到着時にラッチがオープンなためデータパス・タイミングには影響がないことに注目してください．このトランスペアレント・ラッチの性質はARM設計のあちこちで利用され，クロックがクリティカルな信号を遅らせないようになっています．

したがって最小データパス・サイクル時間は，以下の合計になります．

① レジスタ読み取り時間
② シフタ遅延
③ ALU遅延
④ レジスタ書き込みセットアップ時間
⑤ フェーズ2からフェーズ1の非オーバラップ時間

上記の中で，もっとも重要なのはALU遅延です．ALU遅延は実行している処理によって大きく変化します．論理演算はキャリを伝える必要がないので比較的高速です．算術演算(加算，減算，比較)では，キャリがワード幅を伝播することもあるので，長い論理パスが必要になります．



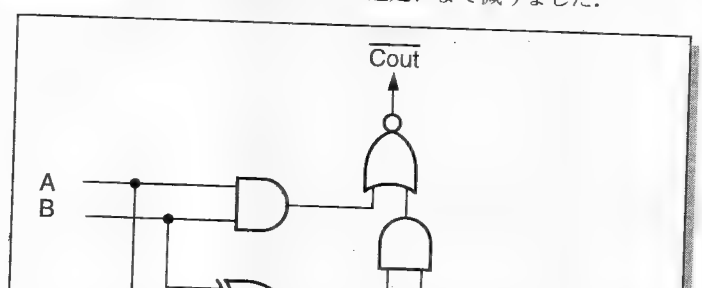
〔図4.9〕ARMデータパス・タイミング(3段パイプライン)

◆加算器の設計

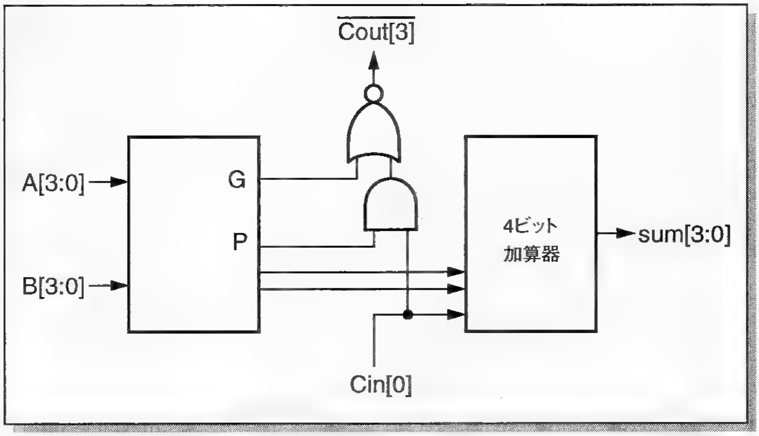
32ビット加算時間はデータパス・サイクル時間に大きく影響し，最大クロック速度やプロセッサ性能にも影響してくるため，ARMの後続バージョンを開発するうえで多大な注意が払われてきました．

初代のARMプロセッサ・プロトタイプは，図4.10のように単純なリプル・キャリ加算器を使っていました．キャリ論理にCMOS AND-OR-INVERTゲートとAND/OR交互論理を用いることにより，偶数ビットは図の回路，奇数ビットは反転した入出力を持ちANDおよびORゲートを交換した二重回路を使っていました．そして悪いときには，キャリ・パスの長さが32ゲートにもなっていました．

クロック速度を向上するために，ARM2は4ビット・キャリ・ルックアヘッド方式を使い，最悪の場合のキャリ・パスの長さを短縮しました．回路は図4.11に示してあります．この論理は，キャリ生成(G)と伝播(P)信号を出して4ビットのキャリ・アウトを制御します．キャリ伝播パスの長さは，やはりAND-OR-INVERTゲートとAND/OR交互論理の組み合わせを使って8ゲート遅延にまで減りました．



〔図4.10〕初代ARM1リプル・キャリ加算器

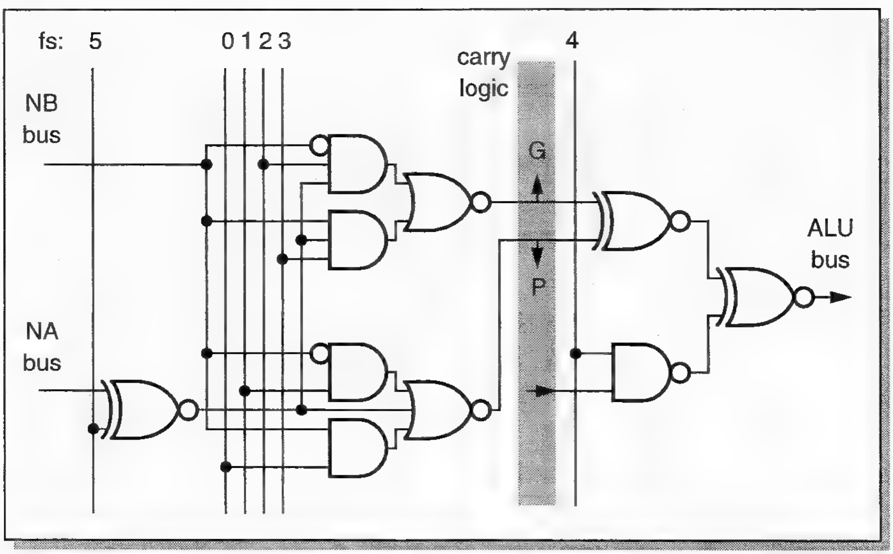


〔図4.11〕ARM2の4ビット・キャリ・ルックアヘッド方式

◆ALU機能

ALUは二つの入力を加算するだけではありません．メモリ転送のためのアドレス計算，分岐計算，ビット単位論理関数など命令セットで定義されたフルセットのデータ処理を実行しなければなりません．

完全なARM2 ALU論理は，図4.12のとおりです．このALUが生成した機能群とALU機能選択の関連値は表4.1にあります．



〔図4.12〕1結果ビットのARM2 ALU論理

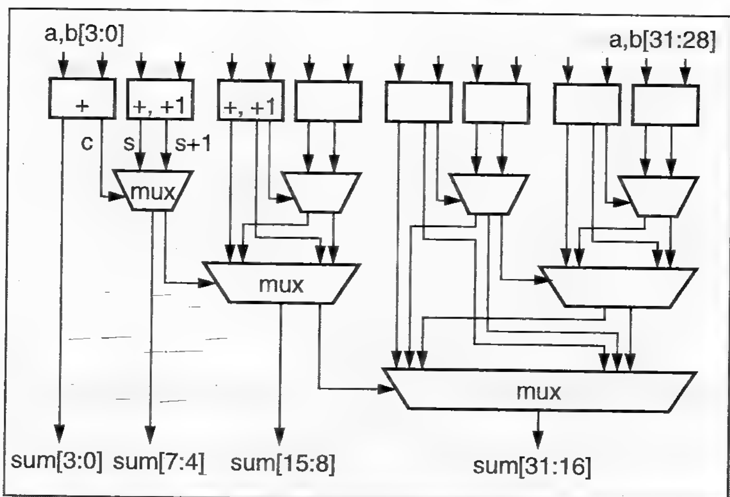
〔表4.1〕ARM2 ALU機能コード

fs5	fs4	fs3	fs2	fs1	fs0	ALU output
0	0	0	1	0	0	A and B
0	0	1	0	0	0	A and not B
0	0	1	0	0	1	A xor B
0	1	1	0	0	1	A plus not B plus carry
0	1	0	1	1	0	A plus B plus carry
1	1	0	1	1	0	not A plus B plus carry
0	0	0	0	0	0	A
0	0	0	0	0	1	A or B
0	0	0	1	0	1	B
0	0	1	0	1	0	not B
0	0	1	1	0	0	zero

◆ARM6キャリ選択加算器

最悪ケースの加算時間は，キャリ選択加算器の使用によりARM6ではさらに向上しました．この加算器は，ワードの様々なフィールドの合計をキャリインが‘0’と‘1’の両方の場合で計算し，最終的な結果は正しいキャリイン値を使って選択し，マルチプレクサを制御します．方式全体は図4.13のとおりです．

クリティカル・パスの長さはlog2［ワード幅］ゲートですが，ゲートのうちいくつかのファンアウトが高いため前の方式との直接的比較は困難です．もっとも，大きな面積を使わなければなりませんが，最悪ケースの加算時間は，4ビット・キャリ・ルックアヘッド加算器よりかなり高速です．



〔図4.13〕ARM6キャリ選択加算器方式

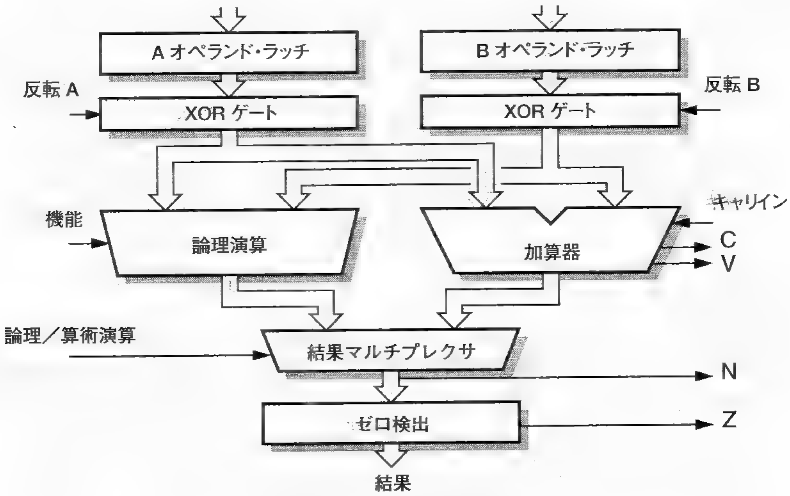
◆ARM6 ALU構造

ARM6のキャリ選択加算器は，ARM2のように，算術演算と論理演算を簡単に一つの構造にまとめることはできません．しかし，別個の論理ユニットが加算器と並行して動作しており，マルチプレクサは必要に応じて

加算器か論理ユニットのどちらかから出力を選択します．

ALU構造の全体は，図4.14のとおりです．入力オペランドはそれぞれ選択的に反転し，加算し論理ユニットで組み合わせます．そして最後に，求める結果を選択してALU結果バスに発行します．

CおよびVフラグ（論理演算においては意味をもたない）は加算器で生成，Nフラグは結果のビット31からコピー，Zフラグは結果バス全体から求めます．Zフラグ作成は32ビットNORゲートを必要とし，これはクリティカル・パス信号になりやすいことに気をつけてください．



〔図4.14〕ARM6 ALU構成

◆キャリ・アービトレーション加算器

加算器論理はARM9TDMIでさらに進歩し「キャリ・アービトレーション加算器」が使用されるようになりました．この加算器は，非常に高速な並列論理構造である「並列プレフィクス」ツリーを使って中間キャリ値をすべて計算します．

キャリ・アービトレーション方式では，従来の伝播生成情報を二つの新しい変数u，vにコーディングしなおします．加算器内の特定のビット位置から入力A，Bを使ってキャリアウトCを計算することを考えてください．キャリが判明する前に得られる情報は，表4.2のとおりです．この表は，同じ情報をu，vを使ってどうエンコードするかも示しています．この情報は，次の公式を使って，隣接するビット位置から得られる情報と組み合わせることができます．

$$(u,v) \bullet (u',v') = (v+u \cdot u', v+u \cdot v') \quad \text{（式12）}$$

この組み合わせ演算子は連想性（associative）があり，通常の並列プレフィクス・ツリーを使えば合計内の全ビットについてu，vを計算することができます．（式12）を実装するのに必要な論理は，一つのCMOSゲートに最大4組の入力を組み合わせることができ，一つのトランジスタ構造から新しいu，v出力を生成します．

さらに，キャリインが1であればuがキャリアウトとなり，キャリインが0であればvがキャリアウトとなることがわかります．したがって，u，vを使えばキャリ・アービトレーション/キャリ選択加算器の組み合わせに必要な値（Sum，Sum+1）を生成することができ，結果として多様な設計が生まれ，性能，領域，電力消費の間のトレードオフを調整するための設計上の選択肢になっています．

〔表4.2〕
ARM9 キャリ・アービトレーションのエンコード

A	B	C	u	v
0	0	0	0	0
0	1	unknown	1	0
1	0	unknown	1	0
1	1	1	1	1

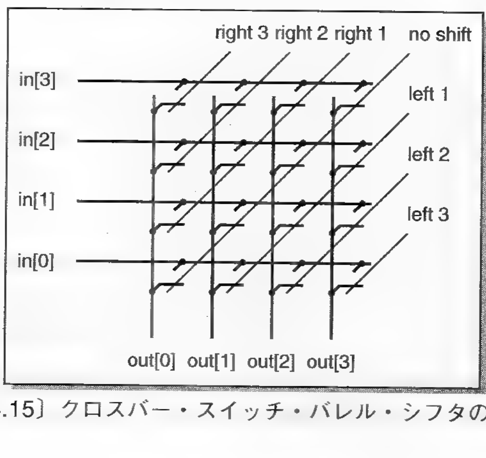
◆バレル・シフタ

ARMアーキテクチャは図4.1で示される構成にあるように，ALU処理と連続してシフト処理を行う命令をもっています．図4.9のデータパス・タイミング図に示すようにシフト時間は直接データパス・サイクル時間に影響するため，シフタ性能はクリティカルです(他のプロセッサ・アーキテクチャは，シフタとALUを並列にもつことが多いため，シフタがALUより遅くない場合はデータパス・サイクル時間に影響しない)．

シフタを通る遅延を最小化するため，クロスバー・スイッチ・マトリックスを使って各入力を適切な出力に導きます．クロスバー・スイッチ・マトリックスの基本は，図4.15の4×4マトリックスのとおりです(ARMプロセッサは32×32マトリックスを使用している)．各入力は，スイッチを通って各出力につながります．ARMデータパスのようにプリチャージ式ダイナミック回路が使われている場合は，各スイッチは1個のNMOSトランジスタで実装できます．

シフト機能は，スイッチを対角線に沿って共通制御入力へつなぐことによって実現します．

(1) 左または右シフト機能では，1本の対角線がオンになる．それによって，全入力ビットが対応の出力へそれぞれ使われている場所でつながる(「はみ出る」ビットもあるので，全部が使われているわけではない)．ARMでは，バレル・シフタは，‘1’は接地電圧に近いことを表し，‘0’は電源電圧に近いことを表す否定論理で動作している．前もって充電しておくことで全出力が論理的‘0’にセットされ，あるスイッチ処理の間にどの入力にも接続していない出力は‘0’のままになり，シフト動作で必要となる‘0’詰めがなされる．
(2) 右回転機能では，右シフト対角線が補数の左シフト対角線と一緒にイネーブルされる．たとえば，4ビット・マトリックスでは，右1ビットは「右1」と「左3」(3 = 4－1)対角線を使って実現する．
(3) 算術右シフトでは，接続していない出力ビットを0で埋めるのではなく符号拡張を使う．シフト量のデコードと適切な出力放電には，別の論理が使われる．



〔図4.15〕 クロスバー・スイッチ・バレル・シフタの原理

◆乗算器の設計

初代のプロトタイプを除いてARMプロセッサはすべて，整数乗算を扱うハードウェアをもっています．乗算器は次の2方式が使われています．

(1) 旧いARMコアは，32ビット結果乗算と積和命令だけを扱う乗算ハードウェアをもつ．

(2) 最近のARMコアは，高性能乗算ハードウェアをもち，64ビット結果乗算と積和命令を扱う．

(1) のほうは，主データパスを反復使用し，バレル・シフタとALUがクロック・サイクルごとに2ビット積を生成します．乗算レジスタに1がなくなるとアーリ・ターミネーション論理によって反復を止めます．

乗算器は，修正ブース（Booth）アルゴリズムを用いて2ビット積を生成します．つまり，×3は，×(-1)+×4として実現できることを利用します．これにより，2ビット乗算器の四つの値は全部，次サイクルに×4を持ち越すことで簡単なシフトと加算または減算によって実現できることになります．

乗算のN番目サイクルの制御設定は，表4.3のとおりです（×2の場合は，減算とキャリを用いても実現できる．キャリなしの加算も使用できるが，制御論理は前記のほうが多少簡単になる）．

この乗算は既存のシフタとALUを使うため，追加しなければならないハードウェアは，乗算器用の専用の2ビット/サイクルのシフト・レジスタとブース・アルゴリズムに使うわずかなゲートだけで済みます．全体でも，ARMコアのうち数%しか占有しません．

〔表4.3〕*N*番目サイクルの2ビット乗算アルゴリズム

キャリイン	乗算器	シフト	ALU	キャリアウト
0	×0	LSL #2*N*	A + 0	0
	×1	LSL #2*N*	A + B	0
	×2	LSL #(2*N* + 1)	A − B	1
	×3	LSL #2*N*	A − B	1
1	×0	LSL #2*N*	A + B	0
	×1	LSL #(2*N* + 1)	A + B	0
	×2	LSL #2*N*	A − B	1
	×3	LSL #2*N*	A + 0	1

◆高速乗算器

乗算性能を重要視すると，ハードウェア・リソースが多く必要になります．いくつかの組み込み型システムでは，ARMコアは一般的な制御機能だけでなくリアルタイム・ディジタル信号処理（DSP）も行うようになっています．DSPプログラムは通常，乗算に重点が置かれ，リアルタイムの制約に合わせるためには乗算ハードウェアの性能がクリティカルになります．

一部のARMコアで使用している高性能乗算器は，広く普及している冗長バイナリ表現を利用し，部分積を加算することからくるキャリ伝播の遅延を防ぎます．中間結果は，部分和と部分キャリとして保持し，本当のバイナリ結果は，主ALU内の加算器など**キャリ伝播加算器**で二つの和を加算して得ます．ただし，これは乗算の終わりで一度しか行いません．乗算の間，部分和とキャリは**キャリ保存加算器**で組み合わせ，そこでキャリは追加ステージごとに1ビットしか伝播しません．したがってキャリ保存加算器は，キャリが32ビット全部を伝播しなければならないキャリ伝播加算器より大幅に短い論理パスをとります．そして，いくつかのキャリ

保存処理は，キャリ伝播処理を1回だけ含み1クロック・サイクルで実行できます．

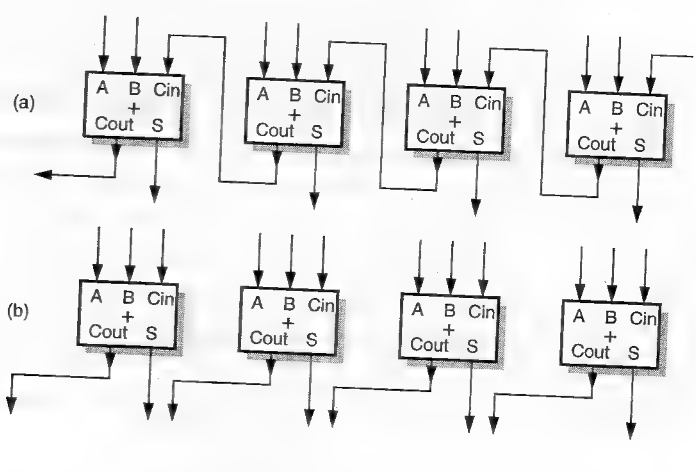
キャリ保存加算器の作り方はたくさんありますが，もっとも単純なのは3入力2出力形式です．これは，入力として，すべて同じバイナリの重みをもつ部分和，部分キャリ，部分積を受け入れ，出力として新しい部分和と，和の2倍の重みをもつ新しい部分キャリを生成します．各ビットの論理機能は，リプル・キャリ式キャリ伝播加算器（図4.10参照）に使われる通常の全加算器と同じですが，構造が異なります．図4.16に，2構造を示します．キャリ伝播加算器は，二つの一般的（非冗長）バイナリ数を入力とし，バイナリ和を生成します．キャリ保存加算器は一つのバイナリ入力と一つの冗長（部分和と部分キャリ）入力で，冗長バイナリ表現の和を生成します．

反復乗算ステージでは，和をフィードバックし，反復ごとに一つの新しい部分積と組み合わせます．すべての部分積を加算したあと，ALUのキャリ伝播加算器で部分積と部分キャリを加算して冗長表現を一般的バイナリ数値に変換します．

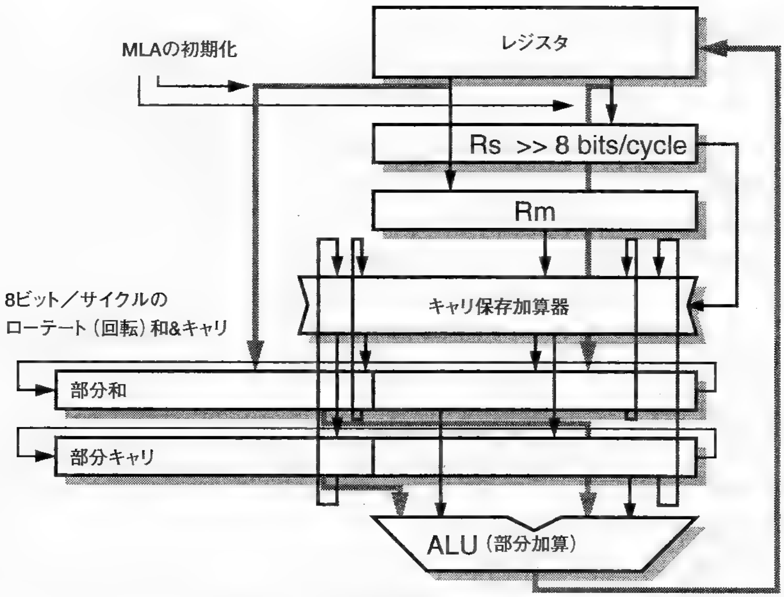
高速乗算器には，数層のキャリ保存加算器が順に並んでいて，それぞれが一つの部分積を処理します．表4.3に類似した形で部分積が修正ブース・アルゴリズムにしたがって生成されれば，キャリ保存加算器の各ステージは各サイクルで乗算器の2ビットを処理します．

いくつかのARMコアで使っている高性能乗算器の全体構造は，図4.17のとおりです．レジスタ名は，5.8節で説明した命令フィールドを指します．キャリ保存アレイは4層の加算器をもち，それぞれが二つの乗算ビットを処理するので，アレイはクロック・サイクルごとに8ビットを乗算できます．命令の最初で部分和およびキャリ・レジスタをクリアするか，もしくは部分和レジスタを累積値に初期化します．乗算器は「R群」レジスタでサイクルごとに8ビット右へシフトしますから，部分和およびキャリはサイクル毎に8ビット右へ回転します．アレイは，乗算器がトップ・ビットに十分なゼロをもつ場合，命令を少ないサイクルで完了するためのアーリ・ターミネーションを使って4回までサイクルします．また，部分和およびキャリは1回に32ビットを組み合わせてレジスタ・バンクに再度書き込みます（乗算が早期に終了する場合，部分和とキャリの再整列が多少必要になる．これは，図4.17には示されていない）．

高速乗算器は，他のARMコアで使っている低コストのソリューションより，かなり多くの専用ハードウェアを必要とします．160ビットシフト・レジスタ，128ビット・キャリ保存加算器論理もあるのです．増加した領域は，ARM8やStrongARMなど高性能コアでは多少小さい割合になるものの，単純なプロセッサ・コアでは10%にもなります．しかし，乗算速度が約3倍になり，乗算命令の64ビット結果形式の追加機能もサポートするのが利点となります．



〔図4.16〕
キャリ伝播加算器(a)と
キャリ保存加算器(b)の構造



〔図4.17〕ARMの高速乗算器構成

◆レジスタ・バンク

ARMデータパスの最後の主要ブロックは，レジスタ・バンクです．ここに，ユーザ可視状態すべてが，31本の汎用32ビット・レジスタにストアされており，全部で1Kビット近くのデータにもなります．基本的な1ビット・レジスタ・セルは設計において何度も繰り返されますから，サイズの最小化に力を入れるのも当然といえます．

ARM6までのARMコアに使用しているレジスタ・セルのトランジスタ回路は，図4.18のとおりです．記憶セルは非対称のCMOSインバータの逆接続対で，レジスタ内容が変更される場合はALUバスからの強い信号によって，オーバドライブされます．フィードバック・インバータの能力は，新しい値に対するセルの抵抗を最小化するため弱くしてあります．AおよびBの読み取りバスは，クロック・サイクルのフェーズ2の間に前もってV_{dd}に充電されていますから，レジスタ・セルは読み取りバスを放電するだけです．これは，読み取りラインがイネーブルされるとn型パス・トランジスタを使って行います．

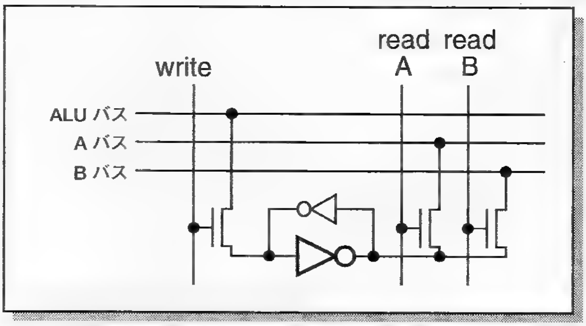
レジスタ・セルの設計は5V電源では容易ですが，n型パス・トランジスタを使って‘1’を書くのは，それ以下の低電源電圧では困難になります．低電源電圧は電力効率を向上するため，ARM6以降のARMはフルCMOS伝送ゲート（書き込み回路のn型パス・トランジスタに並列にp型トランジスタがあり，相補な書き込みイネーブル制御ラインを必要とする）か，より進んだレジスタ回路を使ってきました．

これらのレジスタ・セルは，列状に並んで32ビット・レジスタを形作っており，各列がくっつき合って完全なレジスタ・バンクになっています．読み出しおよび書き込みイネーブル・ラインのデコーダは，図4.19のように列の上にぴったり並んでおり，イネーブルは縦に，データバスは水平に，レジスタ・セルのアレイを走ることになります．デコーダはレジスタ・セル自体よりもっと論理的に複雑であるにもかかわらず，セルに合うよう水平ピッチになっているため，デコーダ配置は非常にきつく，デコーダ自体が長く薄くなければなりません．

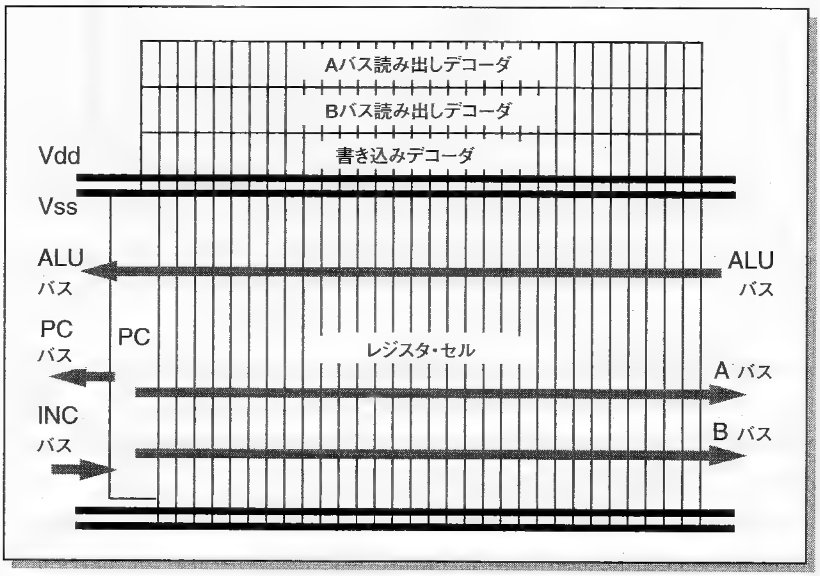
ARMプログラム・カウンタ・レジスタは，単純なコアでは物理的にレジスタ・バンクの一部です．そして，

他のレジスタが一つの書き込みポート，二つの読み出しポートをもつのに対して，二つの書き込みポート，三つの読み出しポートがあります．追加ポートにアクセスでき，「太い」形状を取ることが許される一方の端にプログラム・カウンタを置くことにより，レジスタ・アレイの対称性が保たれています．

レジスタ・バンクは，単純なARMコアではトランジスタ総数の3分の1を占めますが，非常に密度が高く，メモリのような構造をもっているおかげで，占めるシリコン面積は相対的に小さくなっています．二つの読み取りポートをもち，ALUなど複雑な論理機能向けに最適化されたデータパス・ピッチに合わせてあり，SRAMブロックのトランジスタ密度にはおよばないものの，規則性が高く論理機能より密度が高くなっています．



〔図4.18〕ARM2レジスタ・セル回路



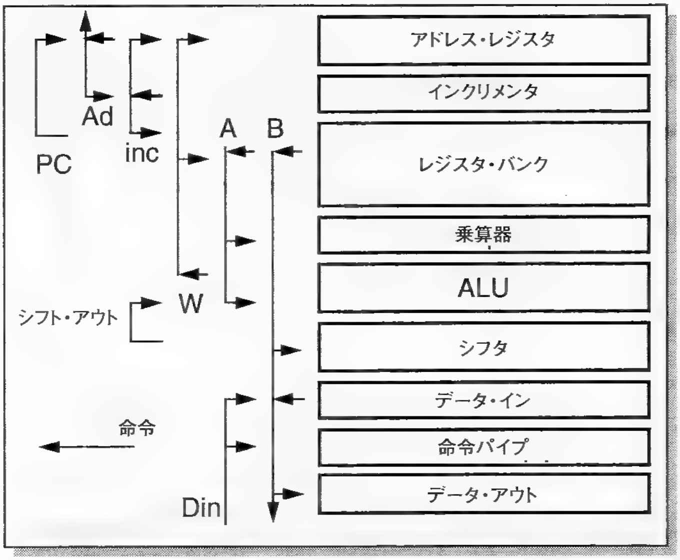
〔図4.19〕ARM2レジスタ・バンク配置図

◆データパス・レイアウト

ARMデータパスは，ビットごとに一定のピッチでレイアウトされています．複雑な機能（ALUなど）に対しては幅広いピッチが最適な一方，単純な機能（バレル・シフタなど）は狭いピッチでレイアウトしたほうが効率的なため，ピッチはその妥協点となります．

そして，各機能はこのピッチでレイアウトされますが，機能を飛び越すバスもありますから（たとえば，Bバスは ALUを通り過ぎるが，ALUに使われることはない）それらの場所も空けておかなければなりません．図4.20のように，各ブロックに載っているバスを記したデータ配置図を書くとよいでしょう．機能ブロックの順序は，複雑な機能を飛び越す追加バスの数を最小にするように決めます．

最近のCMOSプロセスでは，いくつかの金属層に配線することができます（初期のARMコアには二つの金属層があった）．電源や接地，データパスに沿ったバス信号，データパスを横切る制御信号，それぞれに使う配線層は注意して選ぶ必要があります（たとえば，ARM2では，V_{dd}とV_{ss}は金属層2でデータパスの両側に沿って走り，制御ワイヤは金属層1でデータパスを横切り，バスは金属層2でデータパスに沿って並んでいる）．



〔図4.20〕ARMコアのデータパス・バス

◆制御構造

単純なARMコアにおける制御論理には，三つの構造部品があり，図4.21に示すように，相互に関連しています．

(1) 命令デコーダ**PLA**（プログラマブル論理アレイ）．いくらかの命令ビットと内部サイクル・カウンタを使って，次サイクルでデータパス上で実行される処理クラスを定義する．

(2) 主要なデータパス機能ブロックそれぞれに対応する分散型2次制御．主デコーダPLAからのクラス情報を使って，ほかの命令ビットおよび（または）プロセッサ状態情報を選択し，データパスを制御する．

(3) 実行に可変長のサイクルを要する特定の命令（ロードおよびストア乗算，乗算，コプロセッサ処理）向け分散制御ユニット．主デコーダPLAは，リモート制御ユニットが完了を表示するまで一定の状態にロックされる．

主デコーダPLAには14前後の入力，40の積項，40の出力があり，正確な数はコアによって多少変わります．最近のARMコアでは，二つのPLAとして実装しています．つまり，時間クリティカルな出力を生成する小型で高速なPLAと，その他の出力を生成する大型で遅いPLAです．しかし，機能的には一つのPLAとして考えます．

「サイクル・カウント」ブロックは，マルチ・サイクル命令の異なるサイクルを区別し，デコーダPLAが各サイクルで異なる制御出力を生成します．じつはこれは単純なカウンタではなく，不必要なサイクルをとばしたり一定の状態に固定したりする機能をもった汎用ステートマシンなのです．進行中の命令がいつ完了しそうかを見極めて命令パイプラインから次の命令の転送を開始させ，また，条件テストに合わなければ第1サイクルで命令を中止させます．しかし，ほとんどの場合，単純なカウンタと同様の働きをするので，命令サイクル・カウンタと考えても間違いではありません．

◆物理設計

ここまでARMコアの論理設計について主に考えてきて，特定のCMOSプロセスにおける物理的な実装についてはほとんど説明しませんでした．

特定のプロセスでARMプロセッサ・コア（または他のどんなコアでも）を実装するには，二つの基本メカニズムがあります．

ハード・マクロセルが，最終設計に組み込む準備の整った物理レイアウトとして提供されていること．

ソフト・マクロセルが，VHDLなどのハードウェア記述言語で書かれた合成可能設計として提供されていること．

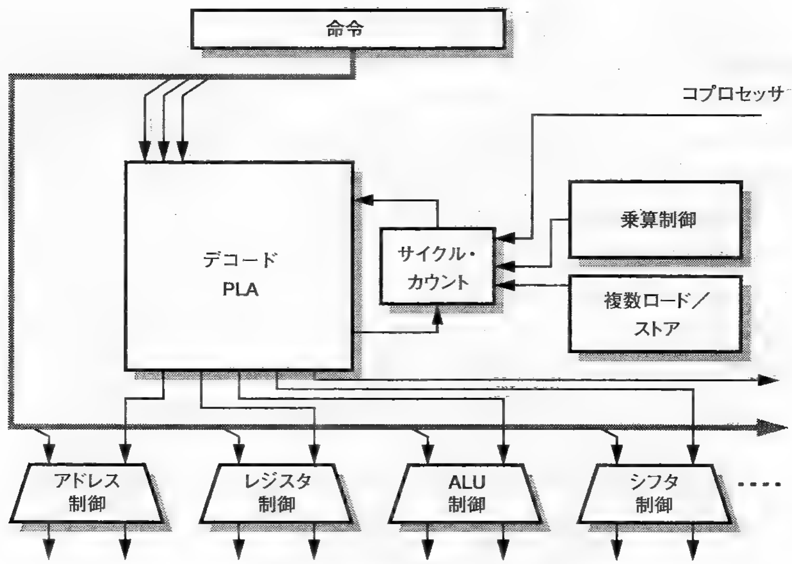
ハード・マクロセルはターゲット・プロセスに完全に合わせこまれていて，フルカスタムでていねいに調整されていることによる面積上の利点があります．しかし，特定のプロセス専用に設計されているため，そのプロセスでしか使えません．新しいプロセスになると，レイアウトを修正して合わせこみをやり直す必要があります．ソフト・マクロセルは新しいプロセス・テクノロジにも簡単に移植できますが，プロセス変更の後はやはり合わせこみをやり直す必要があります．

初期のARMコアは，ハード・マクロセルとしてだけ出荷していました．設計はフルカスタム・データパスに基づき，制御論理は論理図レベルで設計，自動配置配線ツールと標準セル・ライブラリを使ってレイアウトに変換していました．プロセス移植性を高めるため，コアは一般的な設計規則（セル・ライブラリとフルカスタム・データパスの両方について）を使って設計しており，同じ物理レイアウトを幾何学的に変形して，全く同じではないが類似した設計規則をもつさまざまな個別プロセスに同じ物理レイアウトを対応させることが可能でした．

最近のARMコアは，ハード・マクロセルとソフト・マクロセルの両方で出荷されています．ハード・マクロセルは，手作業で描いたフルカスタム・データパスを維持しつつ制御論理に合成を使う傾向にあります．ソフト・マクロセルは，レジスタ転送レベル（RTL）記述から完全に合成可能です．

一部のARMパートナは，その中間を採用しました．新プロセスに移植する基礎としてARMコアのゲート・レベル・ネットリスト記述を使用しているのです．これで移植手順に再合成は必要なくなり，同じネットリスト（自動配置配線ツールを使う）を，新プロセス上に実装した標準セル・ライブラリに対応させるだけになりました．

ハード・マクロセル，ソフト・マクロセル（またはゲート・レベル・ネットリスト）のどちらを使うかは、難しい決断です．ハード・マクロセルは確かにプロセス上における領域，性能，電力効率のどの点でもすぐれていますが，各プロセスに移植するために多大な時間，労力，コストがかかります．ソフト・マクロセルやゲートレベル・ネットリストはもっと柔軟で，自動ツールも今やクオリティが向上して手作業によるレイアウトと同じくらいの性能をもつようになっています．ソフト・マクロセルに移植性があるということは，結局，最新プロセスのソフト・マクロセルか旧プロセスのハード・マクロセルかという選択となりますが，最適化において多少劣っていてもプロセス・テクノロジにおける利点がそれを十分に埋め合わせることもよくあります．



〔図4.21〕ARM制御論理構造

4.5 ARMコプロセッサ・インターフェース

ARMは，ハードウェア・コプロセッサを追加することにより，命令セットの汎用拡張に対応します．また，未定義命令トラップによりコプロセッサのソフトウェア・エミュレーションも可能です．

◆コプロセッサ・アーキテクチャ

コプロセッサ・アーキテクチャは，5.15節で説明します．もっとも重要な特徴は次のとおりです．

(1) 最高16の論理コプロセッサをサポート
(2) 各コプロセッサは，適切なサイズの専用レジスタを最高16本もつことが可能．レジスタは32ビットに限らない．
(3) コプロセッサはロード/ストア・アーキテクチャを用い，レジスタに内部処理を行う命令，メモリから，またはメモリへレジスタをロードおよびストアする命令，ARMレジスタから，またはARMレジスタへデータを転送する命令をもつ．

単純なARMコアは，ボード・レベルでコプロセッサのインターフェースがあるため，コプロセッサは別コンポーネントとして導入できます．速いクロック速度ではボード・レベルのインターフェースは困難なため，高性能ARMではコプロセッサ・インターフェースを特にキャッシュとメモリ管理制御機能向けのオンチップ使用にかぎっていますが，他のオンチップ・コプロセッサをサポートすることもできます．

◆ARM7TDMIコプロセッサ・インターフェース

ARM7TDMIコプロセッサ・インターフェースは「バス・ウォッチング」に基づいています（他のARMコアは異なるテクニックを使用）．コプロセッサは，ARM命令ストリームがARMに流れ込むバスに取り付けてあり，コプロセッサはARM命令パイプラインの動作を真似る内部パイプラインに命令をコピーします．

各コプロセッサ命令が実行を開始するとき，ARMとコプロセッサの間で「ハンドシェイク」があり，実行の準備ができているかどうかを確認します．ハンドシェイクは次の3本の信号を使います．

(1) $\overline{cpi}$ (ARMから全コプロセッサへ)

「CoProcessor Instruction」の略で，ARMがコプロセッサ命令を認識し，実行したいことを示します．

(2) *cpa* (コプロセッサからARMへ)

「CoProcessor Absent」の略で，進行中の命令を実行することができるコプロセッサの不在をARMに告げます．

(3) *cpb* (コプロセッサからARMへ)

「CoProcessor Busy」の略で，コプロセッサがまだ命令の実行を開始できないことをARMに告げます．

タイミング的には，ARMとコプロセッサの両方がそれぞれの信号を自主的に生成しなければなりません．コプロセッサは，*cpa*や*cpb*を生成する前に*cpi*を見るまで待つことはできません．

◆ハンドシェイクの結果

いったんコプロセッサ命令がARMとコプロセッサ・パイプラインに入ると，ハンドシェイク信号により命令の処理方法には四つあります．

(1) 分岐シャドーに入るか，または条件コード・テストに通らないかの理由で，ARMは実行しないと決定する（コプロセッサ命令を含むすべてのARM命令は，条件的に実行される）．ARMは$\overline{cpi}$を発さず，両側とも命令を廃棄する．

(2) ARMは実行を決定する（$\overline{cpi}$を発して，意思表示する）が，存在するコプロセッサがどれも受けられないため*cpa*がアクティブになる．ARMは未定義命令トラップを使い，ソフトウェアを使ってトラップされた命令をエミュレートすることにより復帰する．

(3) ARMは実行を決定し，コプロセッサが承諾するが，まだ実行できない．コプロセッサは*cpa*を“L”にするが，*cpb*は“H”のままである．ARMはコプロセッサが*cpb*を“L”にするまで「ビジー待ち」となり，命令ストリームをこの時点で止める．コプロセッサがビジーの間にイネーブルされた割り込み要求がくると，ARMは割り込みを処理するために待ち状態を打ち切り，コプロセッサ命令は後でリトライする．

(4) ARMは実行を決定しコプロセッサが承諾して直ちに実行する．$\overline{cpi}$，*cpa*，*cpb*のすべてが“L”になり，両側とも命令実行をコミットする．

◆データ転送

命令がコプロセッサ・データ転送命令のとき，ARMが初期メモリ・アドレスを生成します（コプロセッサは，アドレス・バスへの接続を必要としない）が，転送の長さはコプロセッサが決定します．ARMはコプロセッサが完了を表示するまでアドレスをインクリメントし続けます．この目的でも，*cpa*，*cpb*のハンドシェイク信号が使われます．

データ転送は，いったん開始すると割り込みできませんから，コプロセッサはARMの割り込み応答を損なわないよう転送長さを最大16ワードに制限しなければなりません（最大長の複数ロード/ストア命令と同じ）．

◆プリエンプティブ実行

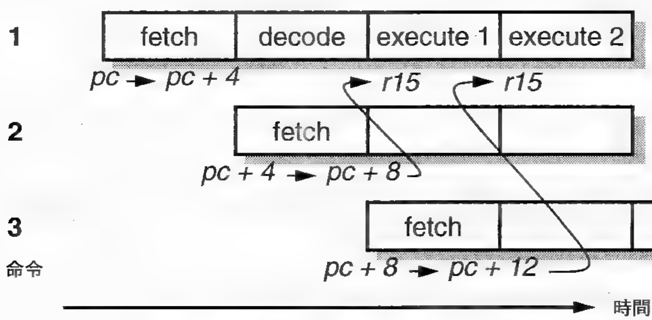
コプロセッサは，ハンドシェイクが最終的に完了しなかった場合にその状態を回復できるのであれば，命令がパイプラインに入った瞬間に実行を開始できます．すべての動作はコミット時点まで**復元可能**（idempotent；反復しても同一の結果を生む）でなければなりません．

4.6 例題と演習

<例題4.1> ARM6において，なぜr15が命令の第1サイクルでpc+8，第2サイクルでpc+12になるのか？

r15は，プログラマに表出したARMパイプラインです．図4.2を振り返ると，いったん現在の命令（図4.22では‘1’）をフェッチし，次（‘2’）をフェッチするとpc値がインクリメントし，第1実行サイクルの始めでpc+8になることがわかります．3番目の命令（‘3’）は第1実行サイクルの間にフェッチし，続くすべての実行サイクルでpc+12になります．

マルチ・サイクル命令はパイプライン・フローを妨げますが，この動作には影響しません．命令はつねに最初の実行サイクルの間に「次の次」の命令をフェッチするため，r15はつねに第1実行サイクルの最初でpc+8から始まり，第2（そしてその次）実行サイクルの最初でpc+12になります（他のARMプロセッサは，この動作をしないことに注意．ARMプログラムを書くときには，あてにしてはいけない）．



〔図4.22〕1サイクル命令のパイプライン処理

演習4.1.1 上のラインにしたがってパイプライン・フロー図を描き，ARM分岐命令のタイミングを図示しなさい（分岐ターゲットは命令の第1実行サイクルで計算し，次のサイクルでメモリへ発行します）．

演習4.1.2 分岐ターゲットが計算された後，分岐ターゲットの命令が実行できるようになるまで，実行サイクルはいくつですか？

プロセッサはこれらの実行サイクルを何に使いますか？

<例題4.2> 図4.10，図4.11に描かれたARM2の4ビット・キャリ論理回路を完成しなさい．

4ビット・キャリ・ルックアヘッド方式は，図4.11の論理で生成した個別のビット・キャリ生成および伝播信号を使います．これらをG[3:0]，P[3:0]のように表示し，4ビット・グループのトップ・ビットからのキャリアウトは次のように求めます．

$$Cout = G[3] + P[3].(G[2] + P[2].(G[1] + P[1].(G[0] + P[0].Cin)))$$

したがって，図4.11で使われているグループ生成および伝播信号G4とP4は，次のように求めます．

$$G4 = G[3] + P[3].(G[2] + P[2].(G[1] + P[1].G[0]))$$
$$P4 = P[3].P[2].P[1].P[0]$$

2信号はキャリイン信号とは独立しているため，その到着に備えてセットアップできます．そして，キャリはAND-OR-INVERTゲート一つの遅延で4ビット・グループに伝播できます．

演習4.2.1　入力と機能選択ラインの観点から，図4.12に示した回路が生成したALU出力1ビットの論理表現を書きなさい．そして，表4.1に列記したALU機能すべてがどのように生成したか示しなさい．

演習4.2.2　リプル・キャリ加算器と4ビット・キャリ・ルックアヘッド加算器について，両設計は図4.12の回路設計を使い，キャリ方式のみ変えて，ゲート数を予測しなさい．

ゲート数の点で，キャリ・ルックアヘッド方式はどのくらい速度が落ちますか？　なぜ，この方式では加算器の規則性，そして設計コストに影響が出るのですか？

第5章 ARM命令セット

第3章では，ユーザ・レベルのARMアセンブリ言語プログラミングを紹介しました．そこで，ARM命令セットの全体像がわかったことと思います．本章では，命令セットの詳細を説明するとともに，標準ARM命令セットに含まれる全命令を見ていきます．

ARMコアは，完全なARM命令のサブセットを16ビット命令にコード化した圧縮形式命令セットを実行できます．これらはThumb(サム：親指)命令で，このThumb命令については，第7章で詳しく解説します．本章ではThumb命令セットのうち，プロセッサをThumb命令実行へスイッチさせる命令に関してのみ触れます．また，一部のARMコアは信号処理機能を向上させるための命令セットの拡張をサポートしています．これについては，8.9節で述べます．

どんなプロセッサ命令セット表でも同じですが，ARM命令セットのなかには，複雑な動作をする隠した部分がいくつかあります．たいていの場合，それらの複雑な動作はプログラマにとってまったく有用ではありません．ARM Limitedではそれらの部分のプロセッサ動作を定義していませんし，プログラマもそれらの部分の命令を使用すべきでありません．ある実装においてARMがある動作をしたからといって，将来の実装で同じ動作をするとは限らないからです．プログラムは意味規則(semantics)によって定義された命令のみを使わなければなりません．

5.1　メモリ構成と特権モード

ARMプログラマ・モデルは，図2.1で紹介しました．本章では，スーパバイザ・モードと例外モードを考えるので，影付きのレジスタが表に出てきます．

◆データ・タイプ

全ARMプロセッサは，次の六つのデータ型に対応します．

(1) 8ビット符号付きおよび符号なしバイト

(2) 2バイト境界で整列(アライン)した16ビット符号付き/符号なしハーフ・ワード

(3) 4バイト境界で整列した32ビット符号付き/符号なしワード

(旧いARMプロセッサの中には，ハーフ・ワードと符号付きバイトをサポートしていないものもある．)

ARM命令はすべて32ビット・ワードで，ワード・アラインされていなければなりません．Thumb命令はハーフ・ワードで(2バイト境界)でアラインしていなければなりません．

注：ARMは腕，その先端にThumb(親指)はある．

ARM内部ではすべて32ビット・オペランドで処理しています．短いデータ型はデータ転送命令でしか対応できません．バイト・データはメモリからロードされるとき，ゼロまたは符号を付加して32ビットに拡張され，内部では32ビット値として処理されます．

ARMコプロセッサは，他のデータ型にも対応しており，とくに浮動小数点値を表す一連のタイプが定義されています．これらはARMコアの中で明示的にサポートされているわけではありませんが，浮動小数点コプロセッサがない場合，上記の標準タイプを使うソフトウェアがこれらのデータ型を解釈します．

◆メモリ構成

バイト・アドレス指定したメモリ内にワード(2バイト)をストアするには，

- 下位バイトを上位バイトより低いアドレスに位置させる：**リトル・エンディアン**
- 下位バイトを上位バイトより高いアドレスに位置させる：**ビッグ・エンディアン**

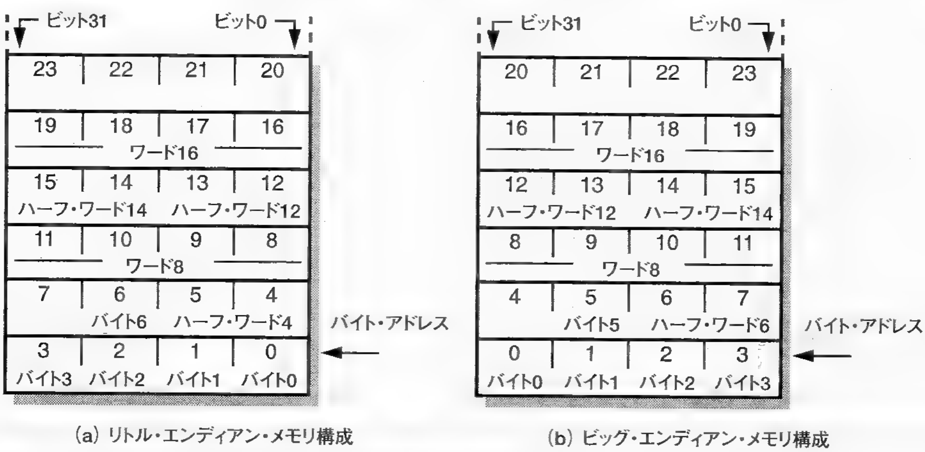
の二つの方法があります．どちらかの方法が優れている，というもっともな理由はありませんので，どちらがよいかという議論は公正な判断というより何を信じるかという問題といえます．

図5.1に，それぞれの方式でデータ型がどのようにストアされているかを示しました(たとえば「ハーフ・ワード12」はアドレス12に位置する)．

「リトル・エンディアン」，「ビッグ・エンディアン」という呼び名は，スウィフトの『ガリバー旅行記』からきたものです．リリプットの住人で小人の一族は卵を細い方の端(リトルエンド)から割るという法律を守っていました．この法律が施行されたとき，住民のうちで卵を太い方(ビッグエンド)から割ることを好む一族が異議を唱え，市民戦争が起こります．ビッグ・エンディアンたちは徐々に対岸の島へ渡り，ブレファスキュー王国を作りました．この市民戦争は多くの死傷者を出しました．実際にコンピュータ・メモリを構成する2方式に「リトル・エンディアン」，「ビッグ・エンディアン」という言葉が使われたのは*Computer*誌，1981年10月号，ダニー・コーエンが著した記事“On Holy Wars and a Plea for Peace”だと思われます．

知っているかぎり，バイトの構成に関する議論で死んだ人はいません(!?)．しかし，異なる方式を使ったマシン間でデータ・セットを転送すると，大きな問題になります．

多くのARMチップは固く中立を守っており，どちらのメモリ構成向けにも設定可能ですが，デフォルトはリトル・エンディアンになっています．本書中では，上位バイトをメモリの上位アドレスにストアするリトル・エンディアン方式を念頭に置きます．ARMは中立かもしれませんが，著者は違うのです！



〔図5.1〕リトル・エンディアン，ビッグ・エンディアンのメモリ構成

◆特権モード

第3章で示したように，多くのプログラムはユーザ・モードで動作します．しかし，ARMには**特権**(privileged)モードもあり，例外やスーパバイザ・コール(**ソフトウェア割り込み**とも呼ばれる)を処理します．

現在の動作モード(operating mode)はCPSR(図2.2参照)の下位5ビットで定義しています．これらのビットの解釈については，表5.1にまとめました．レジスタ・セットがユーザ・レジスタでないときは，図2.1にある適切な影付きレジスタが対応のユーザ・レジスタと入れ替わり，現在のSPSR(Saved Program Status Register；**保存プログラム・ステータス・レジスタ**，次項を参照)もアクセス可能になります．

上記の動作モードすべてには対応しないARMプロセッサもあります．また，旧いARMとの後方互換性をもたせる「26ビット」に対応するものもあります．これについては5.21節で述べます．

特権モードには，制御メカニズムを通してのみ入ることができます．特権モードはきちんとメモリ保護されているため，完全に保護されたOSを作ることも可能です．これについては，第11章で述べます．

多くのARMは，このような保護がされていない組み込み型システムで使用されています．しかし，弱い保護レベルの特権モードにすると，不正なソフトウェアをトラップするのに有効です．

〔表5.1〕
ARMの処理モードとレジスタ用途

CPSR[4:0]	モード	用途	レジスタ
10000	ユーザ	一般ユーザ・コード	user
10001	FIQ	高速割り込み処理	_fiq
10010	IRQ	標準割り込み処理	_irq
10011	SVC	ソフトウェア割り込み処理	_svc
10111	アボート	メモリ・フォールト処理	_abt
11011	未定義	未定義割り込みトラップ・ハンドリング	_und
11111	システム	OSタスクの特権実行	user

◆SPSR

各特権モード(システム・モードを除く)は，SPSR(保存プログラム・ステータス・レジスタ)と関連づけられます．

SPSRは，特権モードに移行した際，ユーザ・プロセスに復帰したとき，ユーザ状態を完全に復元できるように，CPSR(Current Program Status Register；カレント実行プログラム・ステータス・レジスタ)の状態を保存するために用います．

SPSRは，特権モードに入ってからCPSRを復元するのに使うまでは触れられませんが，特権モードのソフトウェアをリエントラントにしなければならないときは(たとえば，スーパバイザ・コードが自分にスーパバイザ・コールを送る場合)，SPSRを汎用レジスタにコピーし，保存しなければなりません．

5.2 例外

例外とは，割り込みやメモリ・フォールトなど，プログラム実行中に起こる予想外のイベントを処理するのに使用します．ARMアーキテクチャでは，ソフトウェア割り込み，未定義命令トラップ(実際には「予想外」

ではありません)，論理的にはプログラム実行中というより実行前に起こるシステム・リセット機能(ただし，プロセッサは実行中に再度リセットできる)も含みます．これらのイベントをすべて「例外」と呼ぶのは，どれもプロセッサ内では同じ基本メカニズムを使うからです．

ARMの例外は三つのグループに分かれます．

(1) 命令実行の直接の影響として起こる例外

ソフトウェア割り込み，未定義命令(要求されたコプロセッサが不在の場合のコプロセッサ命令を含む)，プリフェッチ・アボート(フェッチ中に起こったメモリ・フォールトによって無効になる命令)など．

(2) 命令の付随として起こる例外

データ・アボート(ロードおよびストアのデータ・アクセス中のメモリ・フォールト)．

(3) 命令フローに関係なく外部から起こる例外

リセット，*IRQ*，*FIQ*など．

◆例外エントリ

例外が起こると，ARMは現在の命令をできる範囲まで完了(ただし，リセット例外は現在の命令を即座に終了する)してから，例外処理のために現在の命令シーケンスを離れます．外部イベントの付随として起こる例外エントリは，現在のシーケンス中の次の命令を強引に奪います．直接の影響で起こる例外は，起こる順に処理されます．プロセッサは次の動作シーケンスを行います．

① その例外に対応する動作モードに変わる．
② 例外エントリ命令に続く命令アドレスを新しいモードの`r14`に保存する．
③ `CPSR`の旧い値を新しいモードの`SPSR`に保存する．
④ `CPSR`のビット7をセットして*IRQ*をディセーブルし，さらに，例外が高速割り込みなら`CPSR`のビット6をセットしてそれ以降の高速割り込み*FIQ*をディセーブルする．
⑤ 表5.2にある対応するベクタ・アドレスから実行を開始するようにPCを設定する．

通常，ベクタ・アドレスは対応するルーチンへの分岐を格納しますが，*FIQ*コードは最上位のベクタ・アドレスを占めているので即座に開始できます．

各特権モードにある二つのバンク・レジスタが，リターン・アドレスおよびスタック・ポインタを保持します．スタック・ポインタは，例外ハンドラが使えるように他のユーザ・レジスタを保存するのにも使えます．*FIQ*モードには追加プライベート・レジスタがあり，*FIQ*モードを使うたいていの場合において，ユーザ・レジスタ保存の必要をなくして性能を向上します．

〔表5.2〕
例外ベクタ・アドレス

例外	モード	ベクタ・アドレス
リセット	SVC	0x00000000
未定義命令	UND	0x00000004
ソフトウェア割り込み	SVC	0x00000008
プリフェッチ・アボート(命令フェッチ・メモリ・フォールト)	Abort	0x0000000C
データ・アボート(データ・アクセス・メモリ・フォールト)	Abort	0x00000010
IRQ(通常の割り込み)	IRQ	0x00000018
FIQ(高速割り込み)	FIQ	0x0000001C

◆例外復帰

例外処理が終了すると，ユーザ・タスクは通常，再開します．このときハンドラ・コードは，以下のように，最初に例外が起こったときとまったく同じにユーザ状態を復元しなければなりません．

- 書き換えられたユーザ・レジスタをすべて，ハンドラのスタックから復元する
- CPSRを適切なSPSRから復元する
- PCをユーザ命令ストリームの適切な命令アドレスに戻す

上記のうち最後の二つは，独立して実行できないので注意が必要です．CPSRが先に復元すると，リターン・アドレスを保持しているバンク・レジスタr14へのアクセスが不可能になります．PCが先に復元すると，例外ハンドラは命令ストリームの制御を失い，CPSRの復元が不可能になります．

命令がつねに正しい処理モードでフェッチされ，メモリ保護方式が飛び越されないようにするには，さらに細かい注意が必要です．そこでARMには，上記二つの処理が自動的に一つの命令（の一部）で行えるようにするメカニズムが二つ用意されています．一つは「リターン・アドレスがバンクのr14にあるとき」に使い，もう一つは「リターン・アドレスがスタック上に保存されているとき」に使います．

(1) リターン・アドレスがr14にある場合

まず，リターン・アドレスがr14にある場合を見てみましょう．

- SWIまたは未定義命令トラップから復帰するには次のようにします．

```
        MOVS    pc, r14
```

- *IRQ*，*FIQ*またはプリフェッチ・アボートから復帰するには次のようにします．

```
        SUBS    pc, r14, #4
```

- データ・アボートから復帰，データ・アクセスを再試行するには次のようにします．

```
        SUBS    pc, r14, #8
```

OPコードの後の‘S’修飾子は，デスティネーション・レジスタがPCのとき，特別な命令形式を意味します．リターン命令は，どのような場合にリターン・アドレスを調整するのか見てみましょう．

- IRQとFIQは，例外エントリによって「奪われた」命令を実行するために1命令早く復帰しなければならない．
- プリフェッチ・アボートは，最初に要求されたときにメモリ・フォールトを起こした命令を実行するために1命令早く復帰しなければならない．
- データ・アボートは，データ転送命令，つまり例外エントリによって奪われた命令の前の命令を再試行するために2命令早く復帰しなければならない．

(2) リターン・アドレスがスタック上にある場合

ハンドラがリターン・アドレスをスタック上にコピーしている場合（たとえば，リエントラント動作をするためなど．ただし，この場合はPCとともにSPSRも保存しなければならない）は，ユーザ・レジスタの復元と復帰は，次のように，一つの複数レジスタ転送命令によって実現される．

```
        LDMFD   r13!, {r0-r3,pc}^  ; restore and return
```

レジスタ・リストの後の「^」（PCを含まなければならない）は，これが特別な形式の命令であることを示しています．CPSRはPCがメモリからロードされると同時に復元されますが，レジスタは昇順にロードされるため，PCはメモリから最後に転送されます．

ここで使うスタック・ポインタ(r13)は，特権モードに属するバンク・レジスタです．各特権モードは，独自のスタック・ポインタをもち，システム・スタートアップ時に初期化します．

明らかなことですが，スタック復帰メカニズムを採用できるのは，r14の値がスタック上に保存される前に必要に応じて調整されている場合のみです．

◆例外優先度

複数の例外が同時に発生することがあるので，例外を処理する優先順位を決める必要があります．ARMでは，次の順位となります．

① リセット(最高優先順位)
② データ・アボート
③ *FIQ*
④ *IRQ*
⑤ プリフェッチ・アボート
⑥ SWI，未定義命令(コプロセッサ不在を含む)．これらは相互排他的に命令をエンコードしているので同時に起こることはありえない．

リセットは，プロセッサを既知状態からスタートするため，ペンディングになっている他のすべての例外は無効になります．

もっとも複雑なケースは，FIQ，IRQ，第3番目の例外(リセット以外)が同時に起こるものです．FIQはIRQより優先順位が高くIRQをマスクするので，FIQハンドラが明示的にIRQをイネーブルするかユーザ・コードに戻るまで，IRQは無視されます．

第3番目の例外がデータ・アボートのとき，プロセッサはデータ・アボートハンドラに入り，その後すぐFIQハンドラに入ります．データ・アボートエントリはFIQをマスクしないからです．データ・アボートは復帰パスに「記憶」され，FIQハンドラが復帰すると処理されます．

第3番目の例外がデータ・アボートでないときは，ただちにFIQハンドラに入ります．FIQとIRQが両方完了するとプログラムは第3番目の例外が生成した命令に戻り，そして残りすべてのケースで例外が再帰して処理されます．

◆アドレス例外

表5.2に，最初の8ワードのメモリ位置が0x00000014以外すべて例外ベクタ・アドレスとして示されていることに気づいた読者もいるでしょう．0x00000014は，かつて26ビット・アドレス空間で動作していた初期のARMプロセッサで，アドレス空間外に落ちたロードまたはストア・アドレスをトラップするのに使われていました．これらのトラップは「アドレス例外」と呼んでいました．

32ビットARMは32ビット・アドレス空間外に落ちるアドレスは生成できないため，アドレス例外は現在のアーキテクチャでは不要であり，0x00000014のベクタ・アドレスは使われません．

5.3 条件実行

ARM命令セットのユニークな特徴は，全命令(一部のversion5T命令を除く)が条件付きで実行できることです．条件分岐は多くの命令セットの標準的特徴ですが，ARMは条件実行をスーパバイザ・コールやコプロ

セッサ命令を含むすべての命令に拡張しています．条件フィールドは32ビット命令フィールドの最上位4ビットにあります（図5.2）．



〔図5.2〕ARM条件コード・フィールド

cond:条件コード

条件フィールドにある16の値それぞれが，命令を実行，またはCPSRのN，Z，C，Vフラグの値にしたがってスキップさせます．条件は表5.3に示します．ARMの命令ニモニックも，すべてこの表にあるとおり2文字をつけ加えることで拡張できます．ただし，「always条件（AL）」はデフォルト条件であり，他の条件が指定されないかぎりはそれとみなされますので，省略します．

〔表5.3〕ARMの条件コード

OPコード [31:28]	拡張ニモニック	意味	実行のためのステータス・フラグ
0000	EQ	Equal / equals zero	Z セット
0001	NE	Not equal	Z クリア
0010	CS/HS	Carry set / unsigned higher or same	C セット
0011	CC/LO	Carry clear / unsigned lower	C クリア
0100	MI	Minus / negative	N セット
0101	PL	Plus / positive or zero	N クリア
0110	VS	Overflow	V セット
0111	VC	No overflow	V クリア
1000	HI	Unsigned higher	C セット and Z クリア
1001	LS	Unsigned lower or same	C クリア and Z セット
1010	GE	Signed greater than or equal	N と V は等しい
1011	LT	Signed less than	N と V は等しくない
1100	GT	Signed greater than	Z clear and N equal V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	いずれも
1111	NV	Never (do not use!)	なし

◆never条件

「never（否定）条件（NV）」は使えません．ARMコードでNOP（プロセッサに影響しない命令）を書く方法はほかにたくさんあります．never条件を避ける理由についてARM社は，この命令空間の領域を将来ほかの目的で使う可能性があるから，としています（アーキテクチャversion5Tでは，ほかの目的で使用）．ですから現在のARMが予想どおり動作しても，将来のバージョンが同様に動作するとは保証できません．

◆代替ニモニック

条件表の同じ列にほかの記号も書いてあるときは，条件フィールドを解釈する方法が二つ以上あるというこ

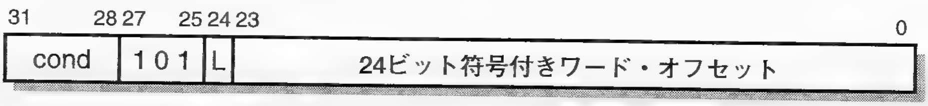
ことです．

たとえば3列目で，同じ条件フィールド値は記号拡張CSまたはHSでも発生します．どちらも，CPSRでCビットがセットされていれば命令を実行します．二つ記号があるのは，同じテストが異なる状況で使われるためです．前に二つの符号なし整数を加算したことがあり，加算からキャリアウトがあったかどうかテストするにはCSを使います．二つの符号なし整数を比較したことがあり，1番目が2番目以上であったかどうかをテストするにはHSを使います．代替ニモニックにより，プログラマは符号なし比較がキャリを同じかそれ以上にセットすることを覚えている必要がなくなります．

2番目の条件が最初の条件の反転であるとき，条件がペアになっていることに気づいた読者もいるでしょう．つまり，どんな条件でも逆の条件が可能なのです(alwaysの例外はneverが使えないため除く)．したがって，条件的命令で「もし…なら…」が実装できるときは，つねに逆の条件の命令を使って「…さもなければ…」を加えることができます．

5.4 分岐およびリンク付き分岐(B，BL)命令

分岐およびリンク付き分岐命令は，命令実行のシーケンスを切り替えるために標準的に使われるものです．ARMは，通常，メモリ中の連続したワード・アドレスから命令を実行し，必要があれば個々の命令をスキップするために条件実行を使います．プログラムがシーケンシャル実行から外れなければならないときは，フロー制御命令を使ってプログラム・カウンタを変更します．特別な状況でこれを行う方法はいくつかありますが，分岐とリンク付き分岐命令が標準的な方法です．



〔図5.3〕分岐とリンク付き分岐命令のバイナリ・コード

<<概要>>

分岐とリンク付き分岐命令によりプロセッサは命令実行を開始しますが，その開始アドレスは，命令が指定した24ビット・オフセットを符号拡張し，左へ2ビット・シフトしてワード・オフセットを作り，それを分岐命令のアドレス＋8バイトのプログラム・カウンタに加えることによって計算します(PCオフセットについては4.1節の最後の項「PCの動作」参照)．アセンブラは通常の状況下では正しいオフセットを計算します．

分岐命令の範囲は＋/－32Mバイトです．

リンク付き分岐は，Lビット(ビット24)がセットされていて，分岐の次の命令アドレスを現在のプロセッサ・モードのリンク・レジスタ(r14)に移動します．これは通常，サブルーチン・コールを実行するのに使いますが，そのとき復帰はリンク・レジスタをPCに再度コピーすることによって起こります．

どちらの命令も条件付き，条件なしで実行できます．

<<書式>>

```
B{L}{<cond>}    <target address>
```

'L' はリンク付き分岐を表します．Lがないとき，リンクなし分岐を生成します．"<cond>" は表5.3の拡張ニモニックに入っているものであり，ない場合は"AL"があることになります．"<target address>" は通

常，アセンブラ・コードのラベルで，アセンブラはオフセットを生成します（オフセットは，ターゲット・アドレスと分岐命令プラス8の間の差である）．

<<例>>

（1）非条件ジャンプ：

```
            B        LABEL    ; unconditional jump ..
            ..
    LABEL   ..                ; .. to here
```

（2）ループ実行10回：

```
            MOV      r0, #10 ; initialize loop counter
    LOOP    ..
            SUBS     r0, #1  ; decrement counter setting CCs
            BNE      LOOP    ; if counter <> 0 repeat loop ..
            ..               ; .. else drop through
```

（3）サブルーチン・コール：

```
            ..
            BL       SUB      ; branch and link to subroutine SUB
            ..                ; return to here
            ..
    SUB     ..                ; subroutine entry point
            MOV      PC, r14  ; return
```

（4）条件サブルーチン・コール

```
    ..
    CMP      r0, #5   ; if r0 < 5
    BLLT     SUB1     ; then call SUB1
    BLGE     SUB2     ; else call SUB2
    ..
```

（注：SUB1が条件コードを変えない場合にかぎり，上記の例が正しく動作する．もしBLLTがサブルーチン・コールを実行すれば，BLGEに復帰するからである．SUB1が条件コードを変えれば，SUB2も実行される．）

<<注意>>

（1）ほかのRISCプロセッサを知っている人は，上記の最初の例で，分岐後，LABELに移動する前にARMが命令を実行すると考えたかもしれない．多くのRISCが採用している遅延分岐モデルに従えばそうなるが，ARMは遅延分岐メカニズムを採用していない．

（2）32ビット・アドレスの始まりまたは終わりを通り越そうとする分岐は，予想しない結果を生むことがあるため避けなくてはならない．

5.5 分岐とeXchange，リンク付き分岐とeXchange(BX，BLX)命令

この二つの命令はThumb(16ビット)命令をサポートするARMチップで有効で，プロセッサがThumb(16ビット)命令を実行するスイッチのメカニズムとして加えられました．同様のThumb命令により，プロセッサは32ビットARM命令にスイッチして戻ります．Thumb命令セットは第7章で説明します．

BLXは，アーキテクチャversion5TをサポートするARMプロセッサでのみ使用可能です．

(1) BX | BLX Rm

31–28	27–6	5	4	3–0
cond	0001001011111111111100	L	1	Rm

(2) BLX label

31–28	27–25	24	23–0
1111	101	H	24ビット符号付きワード・オフセット

〔図5.4〕分岐とexchange命令のバイナリ・コーディング

<<概要>>

最初の書式で，分岐ターゲットはレジスタRmに指定されています．Rmのビット[0]は，CPSRのTビットにコピーし，ビット[31:1]はPCに移動します．

(1) Rm[0]が‘1’のとき

プロセッサはThumb命令を実行するためにスイッチし，最下位ビットをクリアすることによりハーフ・ワード境界でアラインしたRmにあるアドレスから実行を開始します．

(2) Rm[0]が‘0’のとき

プロセッサはARM命令を実行し続け，Rm[1:0]をクリアすることによりワード境界でアラインしたRmにあるアドレスから実行を開始します．

2番目の書式で，分岐ターゲットは，命令が指定する24ビット・オフセットを符号拡張し，左へ二つシフトしてワード・オフセットを作り，さらにそれをBX命令プラス8バイトのアドレスを含むプログラム・カウンタに加算することによって計算したアドレスです(PCオフセットについては「PCの動作」の項を参照)．このアドレスのビット1にHビット(ビット24)も加算すると，奇数のハーフ・ワード・アドレスをターゲット命令に選択することが可能となります．ターゲット命令はつねにThumb命令です(ARM命令をターゲットするには，BLを使用する)．正常な状況下では，アセンブラが正しいオフセットを計算します．分岐命令の範囲は+/−32メガバイトです．

両書式のリンク付き分岐形式(version5Tプロセッサでのみ使用可能なBLX)では，Lビット(ビット5)を最初の書式ではセットしますが，分岐に続く命令のアドレスを現在のプロセッサ・モードのリンク・レジスタ(r14)に移動します．これは通常，Thumbサブルーチンを呼び出す際に復帰アドレスを保存するのに使用します．BXをサブルーチン復帰メカニズムに使用している場合，呼び出し側のルーチンの命令セットもこの復帰アドレスとともに保存することができるため，同じ復帰メカニズムでARMまたはThumbサブルーチンからARMまたはThumbのどちらの呼び出し側にも同様に復帰することができます．書式(1)命令は条件実行の場合も非条件実行の場合もありますが，書式(2)命令は非条件実行されます．

```
1:   B{L}X{<cond>} Rm
2:   BLX<target address>
```

<<書式>>

<target address>は通常，アセンブリ・コード内のラベルです．つまり，アセンブラはオフセット（ターゲット・アドレスと分岐命令プラス8のアドレスの差）を生成して，適切ならHビットをセットします．

<<例>>

（1）非条件ジャンプ

```
BX      r0       ; branch to address in r0,
                 ; enter Thumb state if r0[0] = 1
```

（2）Thumbサブルーチンのコール

```
        CODE32          ;ARM code follows
        ..
        BLX     TSUB    ;call Thumb subroutine
        ..
        CODE16          ;start of Thumb code
TSUB    ..              ;Thumb subroutine
        BX      r14     ;return to ARM code
```

<<注意>>

（1）Thumb命令セットをサポートしていないARMプロセッサは，この命令をトラップし，Thumbのソフトウェア・エミュレーションを実行する場合があります．

（2）ARMアーキテクチャversion5Tを実現しているプロセッサだけが，両書式のBLX命令をサポートします（5.23節参照）．

5.6 ソフトウェア割り込み(SWI)命令

ソフトウェア割り込み命令は，OSの機能呼び出しに使用し「スーパバイザ・コール」とも呼ばれます．この命令でプロセッサは，スーパバイザ・モードに入り，アドレス0x08から命令実行を始めます．

メモリ領域を適切に保護すれば，悪いユーザから完全に保護されたOSをARM上に構築することができます．もっとも，ARMが複数ユーザ向けアプリケーションで使われることはほとんどなく，このレベルの保護が求められることはまれです．

31–28	27–24	23–0
cond	1 1 1 1	24ビット・イミディエート

〔図5.5〕ソフトウェア割り込みのバイナリ・エンコーディング

<<概要>>

24ビット・イミディエート・フィールドは命令処理には影響しませんが，システム・コードにより意味を与

えることができます.

条件が適合すると，標準的なARM例外エントリ・シーケンスを使って命令はスーパバイザ・モードに入ります．プロセッサ動作の詳細は次のとおりです．

(1) SWI後の命令のアドレスをr14_svcに保存する．
(2) CPSRをSPSR_svcに保存する．
(3) スーパバイザ・モードに入り，CPSR[4:0]を10011_2，CPSR[7]を1にセットすることにより*IRQ*(*FIQ*は除く)をディセーブルする．
(4) PCを08_{16}に設定し，そこから命令実行を開始する．

SWI後の命令に戻るには，システム・ルーチンはr14_svcを元のPCに再コピーするだけでなく，SPSR_svcからCPSRを復元しなければなりません．これには，次節で述べる特別形式のデータ処理命令が必要です．

<<書式>>

```
SWI{<cond>}     <24-bit immediate>
```

<<例>>

(1) 文字「A」の出力：

```
MOV     r0, #'A'            ; get 'A' into r0..
SWI     SWI_WriteC          ; .. and print it
```

(2) コールに続く文字列を出力するサブルーチン：

```
        ..
        BL      STROUT                  ; output following message
        =       "Hello World",&0a,&0d,0
        ..                              ; return to here
        ..
STROUT  LDRB    r0, [r14], #1           ; get character
        CMP     r0, #0                  ; check for end marker
        SWINE   SWI_WriteC              ; if not end, print ..
        BNE     STROUT                  ; .. and loop
        ADD     r14, #3                 ; align to next word
        BIC     r14, #3
        MOV     pc, r14                 ; return
```

(3) ユーザ・プログラムを終了し，モニタ・プログラムへ復帰：

```
SWI     SWI_Exit            ; return to monitor
```

<<注意>>

(1) プロセッサがすでにスーパバイザ・モードにあってもSWIは実行できる．ただし，元のリターン・アドレス(r14_svc)とSPSR_svcは保存されている必要がある．でなければ，SWI実行時にこれらのレジスタが上書きされてしまう．
(2) 24ビット・イミディエートの解釈はシステムに依存するが，多くのシステムは文字I/Oなどと同様の基本機能を扱う標準的サブセットをサポートしている．

イミディエートは定数表現としても指定できますが，通常望ましいのはプログラムの最初で要求される呼び出しの名前を宣言(そして値をセット)することです(または，ローカルOSでの値を定義したファイルをイン

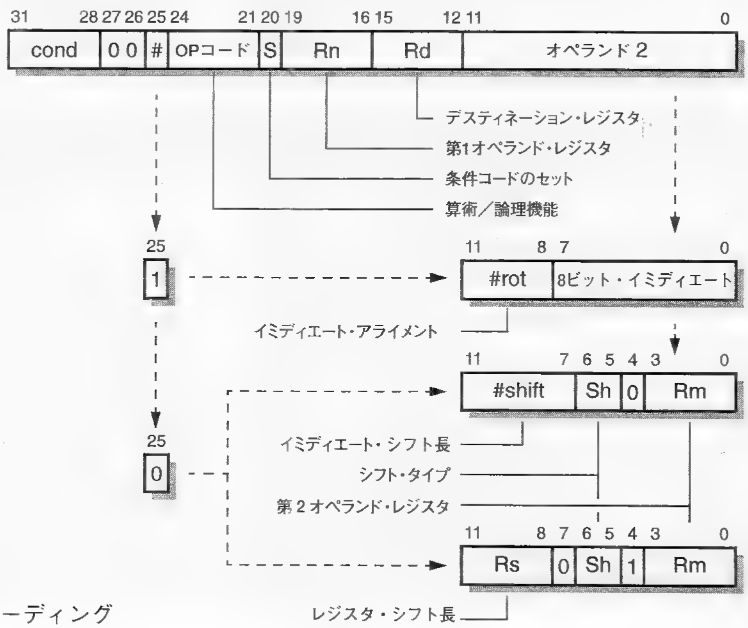
ポート）．そして，その名前をコード内で使います．

(3) 08_{16}にあるスーパバイザ・モードで実行する最初の命令は，メモリ内の近くにあるSWIハンドラへの分岐であることが普通である．SWIハンドラを08_{16}から開始するように書くのは不可能である．なぜなら，$0C_{16}$にある次のメモリ・ワードがプリフェッチ・アボート・ハンドラのエントリ・ポイントだからである．

5.7 データ処理命令

ARMデータ処理命令は，レジスタのデータ値を変更します．対応するのは，32ビット・データ・タイプの演算処理，ビット単位の論理操作などです．オペランドの一つは，ALUへの途中でシフトおよびローテートできるため，1命令でシフトや加算なども可能です．

乗算命令の場合は，異なる形式を使うので，次のセクションで分けて考えます．



〔図5.6〕
データ処理命令の
バイナリ・エンコーディング

<<概要>>

ARMデータ処理命令は，3アドレス形式を採用しています．つまり，二つのソース・オペランドとデスティネーション・レジスタが独立して指定されています．第1ソース・オペランドは，つねにレジスタです．第2のソース・オペランドはレジスタ/シフトしたレジスタ/イミディエート値が可能です．シフトは，第2オペランドに適用され，もしレジスタなら，論理シフト，演算シフトもしくはローテート（図3.1参照）ができ，シフトされる量はイミディエートとして指定されるか第4レジスタによって指定されるかです．

指定されうる処理は表5.4にあります．

命令が可能なオペランドのすべてを要求しないとき（たとえばMOVはRnを，CMPはRdを無視），使われないレジスタ・フィールドはゼロにセットしなければなりません．これは，アセンブラが自動的に行います．

これらの命令は，Sビット（ビット20）を通して，その命令の実行によってプロセッサの条件コードに影響を

与えるかどうかを直接制御できます．クリアされていれば条件コードは変化しません．セットされていると（Rdはr15でないこと），以下のようになります．

- Nフラグは，結果が負のときセットされ，それ以外はクリアされる（つまりNは結果のビット31と等価）．
- Zフラグは，結果がゼロのときセットされ，それ以外はクリアされる．
- Cフラグは，演算処理（ADD，ADC，SUB，SBC，RSB，RSC，CMP，CMN）のとき，ALUからのキャリアウトでセットされ，それ以外はシフタからのキャリアウトでセットされる．シフトが要求されない場合，Cは保存される．
- Vフラグは，演算処理でないとき保存される．演算処理でビット30からビット31へオーバフローがあるときはセットされ，オーバフローがなければクリアされる．演算処理が2の補数の符号付き値と見なされるオペランドをもつとき，Vフラグは意味をもち，結果が範囲を超えていることを示す．

〔表5.4〕ARMデータ処理命令

OPコード [24:21]	ニモニック	意味	効果
0000	AND	Logical bit-wise AND	Rd := Rn AND Op2
0001	EOR	Logical bit-wise exclusive OR	Rd := Rn EOR Op2
0010	SUB	Subtract	Rd := Rn - Op2
0011	RSB	Reverse subtract	Rd := Op2 - Rn
0100	ADD	Add	Rd := Rn + Op2
0101	ADC	Add with carry	Rd := Rn + Op2 + C
0110	SBC	Subtract with carry	Rd := Rn - Op2 + C - 1
0111	RSC	Reverse subtract with carry	Rd := Op2 - Rn + C - 1
1000	TST	Test	Scc on Rn AND Op2
1001	TEQ	Test equivalence	Scc on Rn EOR Op2
1010	CMP	Compare	Scc on Rn - Op2
1011	CMN	Compare negated	Scc on Rn + Op2
1100	ORR	Logical bit-wise OR	Rd := Rn OR Op2
1101	MOV	Move	Rd := Op2
1110	BIC	Bit clear	Rd := Rn AND NOT Op2
1111	MVN	Move negated	Rd := NOT Op2

◆定数による乗算

データ処理命令は，レジスタに小さな定数を掛けるのに使用することがあります．これは，次節で説明する乗算命令を使うよりずっと効率的です．後で例をあげます．

◆r15の使用

PCをソース・オペランドとして使うことができるのは，レジスタ指定のシフト数がないときです．レジスタ指定のシフト数がある場合，三つのソース・レジスタはどれもr15であってはなりません．r15がソース・オペランドとして使われているとき，与えられる値は命令+8バイトのアドレスです（8バイト・オフセットは

プロセッサのパイプライン処理を表す．4.1節の「PCの動作」の項を参照)．

PCは，結果のデスティネーションとして指定されることもあり，その場合，命令は分岐の1形式です．これはサブルーチンから復帰する標準的な方法として使います．

PCがデスティネーション・レジスタRdとして指定されるとき，Sビットは依然CPSRに与える命令の影響力を制御しますが，少し異なる形になります．上記のように，ALU出力に従ってフラグを更新するのではなく，Sビットがセットされていれば現在のモードのSPSRをCPSRにコピーし，たいてい割り込みイネーブル・フラグとプロセッサ動作モードに影響を与えます．このメカニズムはPCとCPSRを自動的に復元し，例外から復帰する標準的な方法でもあります．ユーザ・モードとシステム・モードにはSPSRがないため，この命令が使われることはありません．

<<書式>>

アセンブラ表現は，次のいずれかです．命令が単一オペランド(MOV，MVN)のときはRnを無視し，命令が比較で一つの条件コードのみを出力する比較命令(CMP，CMN，TST，TEQ)のときはRdを無視します．

```
<op>{<cond>}{S} Rd, Rn, #<32-bit immediate>
<op>{<cond>}{S} Rd, Rn, Rm, {<shift>}
```

このとき，<shift>はシフト・タイプ(LSL，LSR，ASL，ASR，ROR，RRX，RRX)と，RRXのとき以外のシフト数，つまり5ビット・イミディエート(#<#shift>)またはレジスタ(Rs)を指定します．

<<例>>

(1) r1をr3に加算，結果をr5におく：

```
ADD     r5, r1, r3
```

(2) r2をデクリメントし，ゼロをチェックする：

```
SUBS    r2, r2, #1          ; dec r2 and set cc
BEQ     LABEL               ; branch if r2 zero
..                          ; .. else fall through
```

(3) r0に5を掛ける：

```
ADD     r0, r0, r0, LSL #2
```

(4) r0に10を掛けるサブルーチン：

```
        MOV     r0, #3
        BL      TIMES10
        ..
TIMES10 MOV     r0, r0, LSL #1     ; x 2
        ADD     r0, r0, r0, LSL #2 ; x 5
        MOV     pc, r14            ; return
```

(5) r0，r1の64ビット整数をr2，r3に加算：

```
ADDS    r2, r2, r0          ; add lower, save carry
ADC     r3, r3, r1          ; add higher and carry
```

<<注意>>

イミディエート・フィールドは32ビット命令のサブセット内でエンコードしなければならないので，32ビット・イミディエート値がすべて表現できるわけではない．図5.6のバイナリ・エンコーディングは，イミディエート値をどのようにエンコードするかを示している．イミディエート値は，8ビット・イミディエート・フィールドを偶数のビット位置までローテートすることによって生成する．

5.8 乗算命令

ARM乗算命令は，レジスタに保持した二つの32ビット・バイナリ数の積を出力します．二つの32ビット・バイナリ数の積は64ビットです．プロセッサの特定バージョンでのみ可能な命令形式の中には，完全な結果を独立して指定した二つのレジスタにストアするものもありますが，ほかでは最下位の32ビットを一つのレジスタにストアするだけです．

31–28	27–24	23–21	20	19–16	15–12	11–8	7–4	3–0
cond	0 0 0 0	mul	S	Rd/RdHi	Rn/RdLo	Rs	1 0 0 1	Rm

〔図5.7〕乗算命令のバイナリ・エンコーディング

<<概要>>

さまざまな乗算形式の機能を表5.5に示しました．表で用いた表記は次のとおりです．

- 「RdHi:RdLo」は，RdHi（最上位の32ビット）とRdLo（最下位32ビット）を連結してできた64ビット数である．「[31:0]」は結果の最下位32ビットしか選択しない．
- 単純な代入は「:=」で表す．
- 累積（右端を左に加える）は「+=」で表す．

Sビットは，他のデータ処理命令と同様，条件コード設定の制御に使用する．

- Nフラグは，32ビット結果を生成する形式では，Rdのビット31の値に，また64ビット結果を生成する形式のときはRdHiのビット31の値にセットされる．
- Zフラグがセットされるのは，RdまたはRdHi，そしてRdLoがゼロのときである．
- Cフラグは意味のない値にセットされる．
- Vフラグは変化しない．

〔表5.5〕乗算命令

OPコード [23:21]	ニモニック	意味	効果
000	MUL	Multiply (32-bit result)	Rd := (Rm * Rs) [31:0]
001	MLA	Multiply-accumulate (32-bit result)	Rd := (Rm * Rs + Rn) [31:0]
100	UMULL	Unsigned multiply long	RdHi:RdLo := Rm * Rs
101	UMLAL	Unsigned multiply-accumulate long	RdHi:RdLo += Rm * Rs
110	SMULL	Signed multiply long	RdHi:RdLo := Rm * Rs
111	SMLAL	Signed multiply-accumulate long	RdHi:RdLo += Rm * Rs

<<書式>>

積の最下位32ビットを生成する命令は次のとおりです.

```
MUL{<cond>}{S}  Rd, Rm, Rs
MLA{<cond>}{S}  Rd, Rm, Rs, Rn
```

次の命令はフル64ビット結果を生成します.

```
<mul>{<cond>}{S}RdHi, RdLo, Rm, Rs
```

ただし, `<mul>`は64ビット乗算タイプ(UMULL, UMLAL, SMULL, SMLAL)のうちのどれか.

<<例>>

二つのベクタのスカラ積を出すには次のようにします.

```
        MOV     r11, #20            ; initialize loop counter
        MOV     r10, #0             ; initialize total
LOOP    LDR     r0, [r8], #4        ; get first component..
        LDR     r1, [r9], #4        ; ..and second
        MLA     r10, r0, r1, r10    ; accumulate product
        SUBS    r11, r11, #1        ; decrement loop counter
        BNE     LOOP
```

<<注意>>

(1) オペランドや結果レジスタのどれにおいてもr15を指定してはならない. そうでなければ予想外の結果を生むことになる.

(2) Rd, RdHi, RdLoは, Rmと異なっていなければならない. また, RdHiとRdLoは同じレジスタであってはならない.

(3) いくつかのARMプロセッサは32ビット乗算命令(MUL, MLA)しかサポートしていない. 64ビット乗算は, 名前に「M」のつくARM7バージョン(ARM7DM, ARM7TDMIなど)とその後続のプロセッサでのみ可能である.

5.9 カウント・リーディング・ゼロ (CLZ_アーキテクチャversion5Tのみ)

この命令は, アーキテクチャversion5TをサポートするARMプロセッサでのみ使用します. この命令は, 数を再正規化するのに有用であり, 他のARM命令を使うよりずっと効率的に機能を果たします.

31 28	27 16	15 12	11 4	3 0
cond	000101100000	Rd	00000001	Rm
		デスティネーション・レジスタ		ソース・レジスタ

〔図5.8〕カウント・リーディング・ゼロ命令のバイナリ・エンコーディング

<<概要>>

この命令は, Rm中の最上位の1のビット位置の番号にRdをセットします. Rmが0の場合, Rdは32にセットされ

ます.

<<書式>>

```
CLZ{<cond>} Rd, Rm
```

<<例>>

```
MOV     r0, #&100
CLZ     r1, r0          ; r1 := 8
```

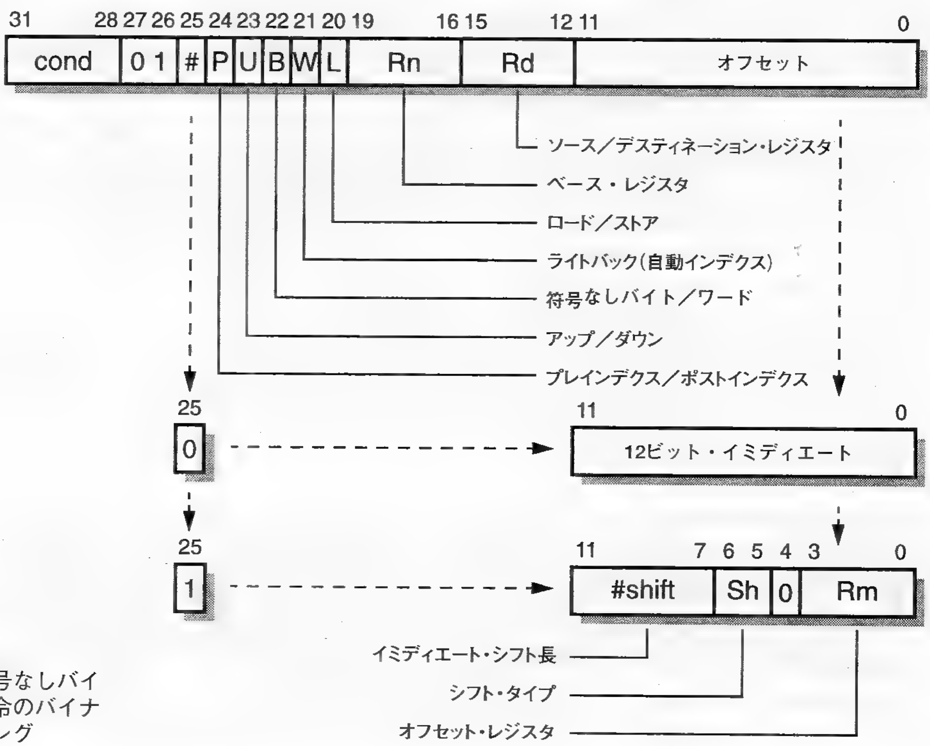
<<注意>>

(1) ARMアーキテクチャversion5Tを実装したプロセッサだけがCLZ命令をサポートしています(5.23節参照).

5.10 1ワード&符号なしバイトのデータ転送命令

1バイトまたは1ワードのデータをARMレジスタとメモリ間で転送するもっとも柔軟な命令です．大きいブロックのデータを転送するには，複数レジスタ転送命令のほうが有効で，またハーフ・ワードや符号付きバイトの転送をサポートするARMプロセッサもあります(ARM7TDMIなど).

レジスタが要求されるメモリ・アドレスの近く(通常は4Kバイト以内)を指すように初期化されている場合，効率的にロード，ストアができ，イミディエートやレジスタ・オフセット，自動インデクス，PC相対など多くのアドレシング・モードも可能です.



〔図5.9〕
1ワードおよび符号なしバイト・データ転送命令のバイナリ・エンコーディング

<<概要>>

この命令は，ベース・レジスタ(Rn)をはじめとして，符号なしイミディエートまたは(スケーリングされた)レジスタ・オフセットを加算(U=1)または減算(U=0)することによって，アドレスを構築します．そして，ベース・アドレスや計算したアドレスを使って，符号なしバイト(B=1)やワード(B=0)量を，レジスタ(Rd)やメモリへ，もしくはレジスタ(Rd)やメモリから，ロード(L=1)したりストア(L=0)したりします．1バイトはレジスタにロードされると32ビットへゼロ拡張します．1バイトがメモリにストアされると，レジスタの下位8ビットがアドレス指定された位置にストアされます．

プレインデクス・アドレシング・モード(P=1)は，計算したアドレスをロードまたはストア処理に使用後，ライトバックが要求された時点で(W=1)ベース・レジスタを計算した値に更新します．

ポストインデクス・アドレシング・モード(P=0)は，修正していないベース・レジスタを転送に使用後，Wビットに関係なく(オフセットは，ベース・レジスタの修飾子と何ら違いはなく，変更の必要がなければいつでもイミディエート・ゼロにセットすることが可能だからである)，ベース・レジスタを計算した値に更新します．Wビットは使用しないため，代わりにユーザ・モードで動作していないコードでのみ動作する機能をもちます．つまりW=1をセットするとプロセッサがメモリへのユーザ・モード・アクセスを要求し，OSはメモリ変換および保護機構のユーザ・ビューを得ることができます．

<<書式>>

(1) プレインデクス形式：

```
LDR|STR{<cond>}{B} Rd, [Rn, <offset>]{!}
```

(2) ポストインデクス形式：

```
LDR|STR{<cond>}{B}{T} Rd, [Rn], <offset>
```

(3) アセンブラに全部まかせる便利なPC相対形式：

```
LDR|STR{<cond>}{B} Rd, LABEL
```

LDRは「ロード・レジスタ」，STRは「ストア・レジスタ」，オプションの'B'は符号なしバイト転送を選択，デフォルトはワードです．<offset>は#+/-<12-bit immediate>または+/-Rm{,shift}が可能で，シフト指定子は，データ処理命令の場合と同じです．ただし，レジスタ指定のシフト量はありません．!は，プレインデクス形式でライトバック(自動インデクス)を選択します．

Tフラグは，メモリ変換および保護システムのユーザ・ビューを選択し，非ユーザ・モードでのみ使います．ユーザはプロセッサが使用されているメモリ管理環境を完全に理解しなければならないため，事実上これはOSに精通している人向けの機能だといえます．

<<例>>

r0のバイトを周辺装置にストアするには：

```
          LDR     r1, UARTADD         ; UART address into r1
          STRB    r0, [r1]            ; store data to UART
          ..
UARTADD &         &1000000            ; address literal
```

アセンブラは，プレインデクス，PC相対アドレシング・モードを使ってアドレスをr1へロードします．これには，リテラルが範囲内(ロード命令から4Kバイト以内)でなければなりません．

<<注意>>

(1) PCをベース・アドレスとして使用すると，命令+8バイトのアドレスが得られる．つまり，オフセット・レジスタとして使用したり，自動インデクス・アドレシング・モード(ポストインデクス・モードも含む)と一緒に使用したりするべきではない．

(2) ワードをPCにロードすることは，ロードしたアドレスに分岐を起こすので，ジャンプ・テーブルの実装の代表的な方法である．バイトをPCにロードするのは避けなくてはならない．

(3) PCをメモリにストアすると，プロセッサの実装により違った結果が出るため，可能なかぎり避ける．

(4) 通常，Rd，Rn，Rmは，互いに異なるレジスタでなければならない．ただし，ベース・レジスタにロードするのは(Rd=Rn)，自動インデクスが同じ命令内で使用されない場合にかぎり可能である．

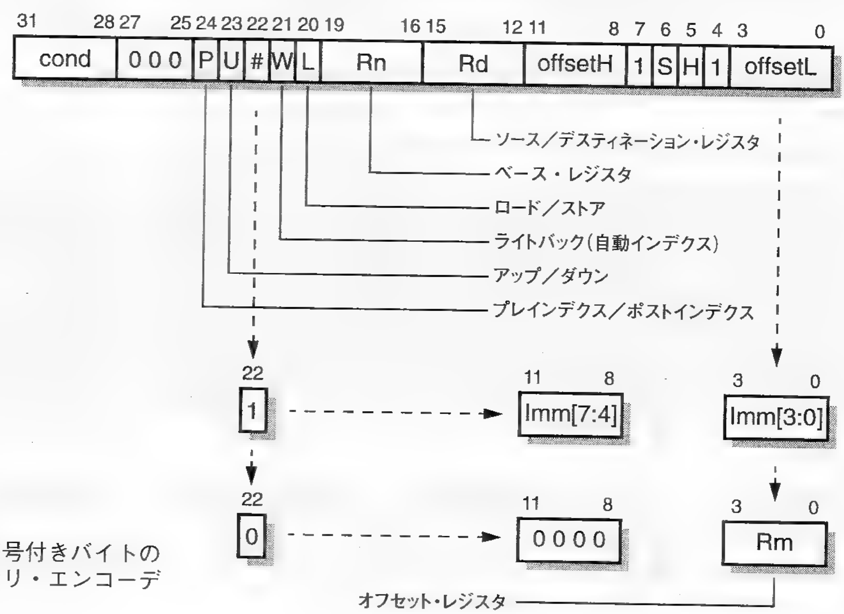
(5) ワード・アラインされていないアドレスからワードがロードされている場合，ロードされるデータはアドレス・バイトを含むワード・アラインされたワードであり，アドレスされたバイトがデスティネーション・レジスタの最下位バイトに入るようローテートされる．この状況で例外を起こすARMシステムもある(CP15レジスタ1のビット1にあるAフラグに制御されるもの．11.2節で説明する)．

(6) ワード・アラインされていないアドレスにワードがストアされる場合，アドレスの下位2ビットは無視され，ワードはそれらがゼロであるようなアドレスにストアされる．この状況で例外を起こすARMシステムもある(やはりCP15レジスタ1のビット1にあるAフラグに制御される)．

5.11 ハーフ・ワード&符号付きバイトのデータ転送命令

初期のARMプロセッサの中には，この種の命令をサポートしていないものもあります．アーキテクチャに最近加えられたため，イミディエート・フィールドが分割しているなど，命令空間に「押し込められて」います．

可能なアドレシング・モードは，符号なしバイトおよびワードの場合に使えるもののサブセットです．



〔図5.10〕
ハーフ・ワードおよび符号付きバイトのデータ転送命令のバイナリ・エンコーディング

<<概要>>

この命令は，前節で説明したワードおよび符号なしバイトに関する命令と非常に似ています．ただし，イミディエート・オフセットは8ビットにかぎり，スケーリングしたレジスタ・オフセットはもはや使えません．

表5.6に示すように，SとHビットで転送するオペランド・タイプを定義します．データ・タイプが符号なしバイトになる組み合わせがないことに注目してください．代わりに前節で説明した書式を使います．ストアするのは符号付きデータでも符号なしデータでも同じですから，重要なのは次の命令書式だけです．

- 符号付きバイト，符号付きハーフ・ワードまたは符号なしハーフ・ワードをロードする
- ハーフ・ワードをストアする

符号なし値はロードされるとき32ビットにゼロ拡張されています．符号付き値はデータのMSBをコピーすることで32ビットに拡張されます．

〔表5.6〕データ・タイプ

S	H	データ・タイプ
1	0	符号付きバイト
0	1	符号なしハーフ・ワード
1	1	符号付きハーフ・ワード

<<書式>>

(1) プレインデクス形式：

```
LDR|STR{<cond>}H|SH|SB Rd, [Rn, <offset>]{!}
```

(2) ポストインデクス形式：

```
LDR|STR{<cond>}H|SH|SB Rd, [Rn], <offset>
```

<offset>は#+/-<-8bit immediate>または+/-Rm，そしてH|SH|SBがデータ・タイプを選択します．それ以外の場合は，ワードおよび符号なしバイト転送の場合と同じ書式です．

<<例>>

符号付きハーフ・ワードの配列をワード配列に拡張するには，次のようにします．

```
        ADR     r1, ARRAY1      ; half-word array start
        ADR     r2, ARRAY2      ; word array start
        ADR     r3, ENDARR1     ; ARRAY1 end + 2
LOOP    LDRSH   r0, [r1], #2    ; get signed half-word
        STR     r0, [r2], #4    ; save word
        CMP     r1, r3          ; check for end of array
        BLT     LOOP            ; if not finished, loop
```

<<注意>>

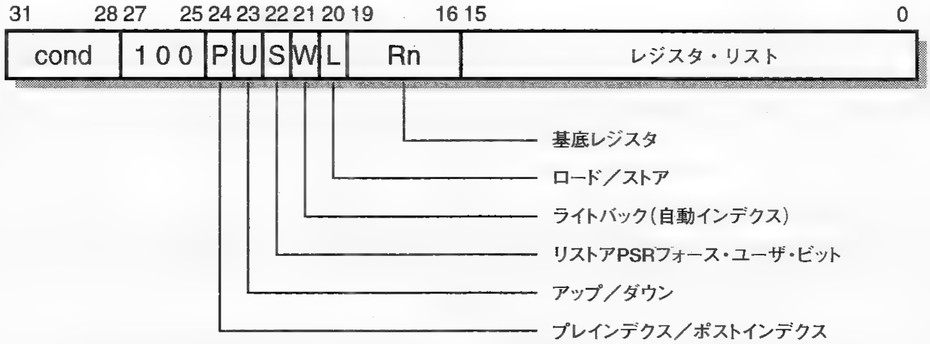
(1) r15とレジスタ・オペランドの使用に関しては，前節で説明したワードおよび符号なしバイト転送と同様の制限がある．

(2) ハーフ・ワード転送には，ハーフ・ワード・アラインのアドレスを使用する．

5.12　複数レジスタ転送命令

ARM複数レジスタ転送命令は，現在の処理モードで可視の16本のレジスタ中，任意（またはすべて）のサブセットをメモリからロード，またはメモリへストアします．また，ある形式を使えば，OSがユーザ・モード・レジスタをロードまたはストアしてユーザ処理状態を保存または復元し，またある形式では，例外ハンドラからの復帰の一環としてSPSRからCPSRを復元します．

複数レジスタ転送命令は，プロシージャ・エントリおよびワーク・スペース・レジスタを保存，復元するための復帰に用い，また，広帯域幅メモリ・ブロック・コピー・ルーチンに使います．



〔図5.11〕複数レジスタ・データ転送命令のバイナリ・エンコーディング

<<概要>>

命令の下位16ビットにあるレジスタ・リストには各可視レジスタ用のビットがあり，ビット0はr0が転送されるか否かを制御し，ビット1はr1を制御，以下PC転送を制御するビット15まで同様に続きます．

レジスタは，ベース・レジスタとアドレシング・モードにより定義されたメモリ・ワード隣接ブロックからロードされたり，そこへストアされたりします．ベース・レジスタは，各ワード転送の前（P=1）または後（P=0）で，インクリメント（U=1）またはデクリメント（U=0）します．自動インデクスもサポートしています．W=1のとき，ベース・レジスタは，命令完了後，転送バイト数だけ増加（U=1）または減少（U=0）します．

特別な命令形式を使えばCPSRを復元できます．PCが複数ロードのレジスタ・リストにあり，Sビットがセットされているとき，現在のモードのSPSRはCPSRにコピーされ，不可分な復帰および状態復元命令を出します．ただしユーザ・モードにはSPSRがないので，ユーザ・モード・コードでは使えません．

PCがレジスタ・リストになく，Sビットがセットされているときは，非ユーザ・モードで実行した複数ロードおよびストアの両命令がユーザ・モード・レジスタを転送します（現在のモードのベース・レジスタを使用）．これにより，OSはユーザ動作状態を保存，復元できます．

<<書式>>

(1) 通常の場合：

```
LDM|STM{<cond>}<add mode> Rn{!}, <registers>
```

<add mode>は，表3.1に示したアドレシング・モードの一つを指定します．命令ビットはこの表に示したメカニズム図に密接に対応しています．「インクリメント」はU=1，「前」はU=0に対応し，「!」は自動インデク

ス(W=1)を指定，<registers>は{ }内のレジスタ・リストとレジスタ範囲です．例：{r0, r3 - r7, pc}

(2) 非ユーザ・モードの場合：

この場合は，CPSRは次によって復元します．

```
LDM{<cond>}<add mode> Rn{!}, <registers + pc>^
```

レジスタ・リストは，必ずPCを含みます．非ユーザ・モードでは，ユーザ・レジスタは次によって保存，復元します．

```
LDM|STM {<cond>} <add mode> Rn,<registers — pc>^
```

このとき，レジスタ・リストはPCを含まず，ライトバックは不可能です．

<<例>>

(1) サブルーチンに入るときに三つのワーク・レジスタとリターン・アドレスを保存する．

```
STMFD   r13!, {r0-r2, r14}
```

このとき，r13はスタック・ポインタとして使用するために初期化されているものとします．

(2) ワーク・レジスタを復元して復帰する．

```
LDMEA   r13!, {r0-r2, pc}
```

<<注意>>

(1) 複数ストア命令でレジスタ・リストにPCが指定されているとき，保存した値は実装に依存する．通常，PCをSTMで指定することは避けたほうがよい(PCをロードすることで期待する結果が得られ，それが手続きから復帰する標準的な方法である)．

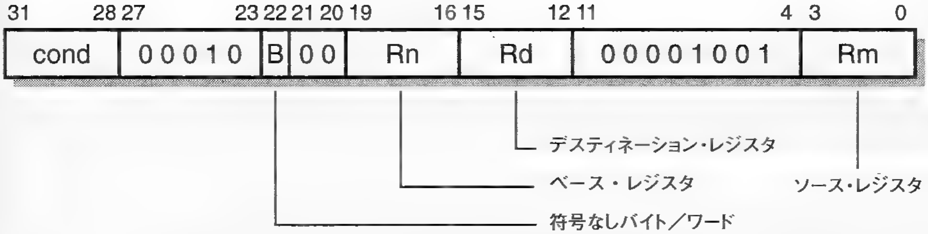
(2) ベース・レジスタは，複数ロードまたは複数ストア命令の転送リストでベース・レジスタを指定できますが，ライトバックを同じ命令内で指定することは予想外の結果を生むことがあるので避ける．

(3) ベース・レジスタがワード・アラインしていないアドレスを含むとき，下位ビットは無視される．ARMシステムの中には例外を生成するものがある．

(4) アーキテクチャversion5Tの場合だけは，ロードしたPCの最下位ビットがThumbビットを更新します．

5.13 メモリとレジスタ間のスワップ命令(SWP)

スワップ命令は，一つの命令内でワードまたは符号なしバイトのロードとストアを組み合わせます．通常，二つの転送を一つの不可分なメモリ操作に組み合わせ，外部メモリ・アクセス(たとえばDMAコントローラから)では分割できなくなります．したがって，命令はセマフォ・メカニズムの基礎として使用でき，複数のプロセス，複数のプロセッサ，プロセッサとDMAコントローラなどで共有するデータ構造への相互排他アクセスが可能になります．このスワップ命令は，セマフォ構築以外ではほとんど使用されません．



〔図5.12〕メモリとレジスタのスワップ命令のバイナリ・エンコーディング

<<概要>>

この命令により，Rnがアドレスしたメモリ位置にあるワード(B=0)または符号なしバイト(B=1)をRdにロードし，Rmからの同じデータ・タイプを同じメモリ位置にストアします．RdとRmは，同じレジスタの場合もあり(ただし，Rnとは異なっていなければなりません)，その場合はレジスタとメモリ値を交換します．ARMはメモリ読み出しとメモリ書き込みを別々のサイクルで実行しますが，2サイクルが分割されないことをメモリ・システムに表示する「ロック」信号を出します．

<<書式>>

```
SWP{<cond>}{B} Rd, Rm, [Rn]
```

<<例>>

```
ADR     r0, SEMAPHORE
SWPB    r1, r1, [r0]      ; exchange byte
```

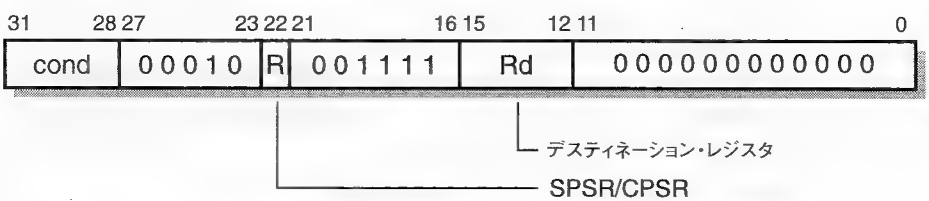
<<注意>>

(1) PCは，この命令中のどのレジスタとしても使用してはならない．

(2) ベース・レジスタ(Rn)は，ソース(Rm)またはデスティネーション(Rn)レジスタと同じではいけない．

5.14 ステータス・レジスタから汎用レジスタへの転送命令

CPSRまたは現在のモードにおけるSPSRの内容を保存または修正する場合，この内容はまず汎用レジスタに転送，選択したビットを変更し，それからステータス・レジスタに値を戻します．このシーケンスの最初のステップを行うのがこの命令です．



〔図5.13〕ステータス・レジスタから汎用レジスタへの転送命令エンコーディング

<<概要>>

CPSR(R=0)または現在のモードのSPSR(R=1)をデスティネーション・レジスタ(Rd)にコピーします．32

ビットすべてをコピーします.

<<書式>>

```
MRS{<cond>} Rd, CPSR|SPSR
```

<<例>>

```
MRS     r0, CPSR           ; move the CPSR to r0
MRS     r3, SPSR           ; move the SPSR to r3
```

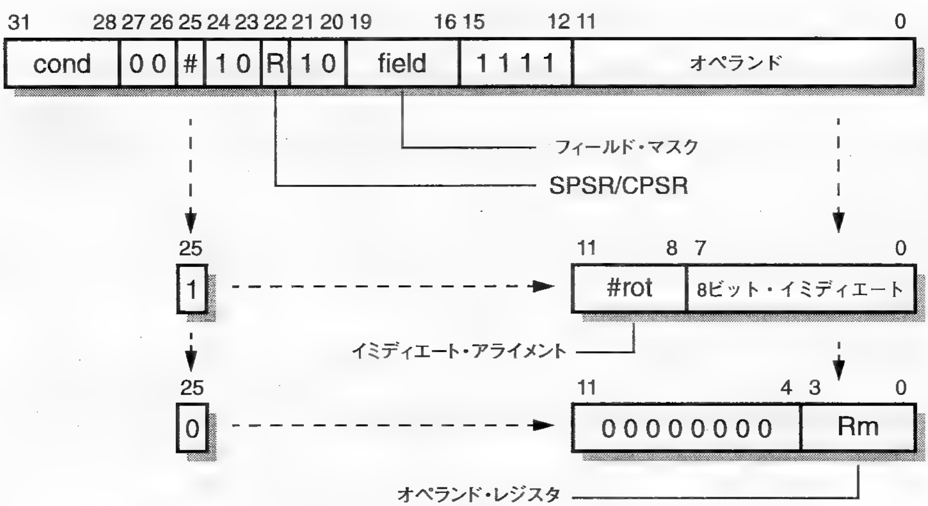
<<注意>>

(1) アクセス可能なSPSRがユーザ・モードやシステム・モードにないので,これらのモードでは命令のSPSR形式を使用してはいけない.

(2) CPSRやSPSRを変更するときは,使用しないビットの値をすべて保つよう注意する.これにより,それらのビットを将来使用する際の互換性をもつ可能性が最高になる.ステータス・レジスタを汎用レジスタに移動し(この転送命令を使って),必要なビットだけを変更し,結果をステータス・レジスタに戻すのがもっとも良い方法である.

5.15 汎用レジスタからステータス・レジスタへの転送命令

CPSRまたは現在のモードにおけるSPSRの内容を保存または変更する場合,この内容はまず汎用レジスタに転送,選択したビットを変更し,それからステータス・レジスタに値を戻します.このシーケンスの最後のステップを行うのがこの命令です.



〔図5.14〕ステータス・レジスタへの転送命令エンコーディング

<<概要>>

オペランドはレジスタ(Rm)またはローテートした8ビット・イミディエート(データ処理命令でオペランド2のイミディエート形式と同様に指定)が可能です.このオペランドをフィールド・マスクのもとでCPSR(R=0)

または現在のモードのSPSR(R=1)へ移動します．

フィールド・マスクがPSRレジスタ内で4バイト・フィールドの更新を制御します．命令のビット16がPSR[7:0]，ビット17がPSR[15:8]，ビット18がPSR[23:16]，ビット19がPSR[31:24]を更新するかどうか決定します．

イミディエート・オペランドを使用するときは，フラグ(PSR[31:24])のみを更新できます(これらのビットだけがユーザ・モード・コードで更新できる)．

<<書式>>

```
MSR{<cond>} CPSR_f|SPSR_f, #<32-bit immediate>
MSR{<cond>} CPSR_<field>|SPSR_<field>, Rm
```

このとき，<field>は以下のうちいずれかです．

- c：制御フィールド-PSR[7:0]
- x：拡張フィールド-PSR[15:8](現在のARMでは不使用)
- s：ステータス・フィールド-PSR[23:16](現在のARMでは不使用)
- f：フラグ・フィールド-PSR[31:24]

<<例>>

(1) N，Z，C，Vフラグをセット：

```
MSR     CPSR_f, #&f0000000; set all the flags
```

(2) N，Z，Vをそのまま保ち，Cフラグだけセット：

```
MRS     r0, CPSR            ; move the CPSR to r0
ORR     r0, r0, #&20000000; set bit 29 of r0
MSR     CPSR_f, r0          ; move back to CPSR
```

(3) スーパバイザ・モードからIRQモードへスイッチ(たとえば，スタートアップでIRQスタック・ポインタを初期化するため)：

```
MRS     r0, CPSR            ; move the CPSR to r0
BIC     r0, r0, #&1f        ; clear the bottom 5 bits
ORR     r0, r0, #&12        ; set the bits to IRQ mode
MSR     CPSR_c, r0          ; move back to CPSR
```

このとき，割り込みイネーブル設定を変えないように元のCPSR値をコピーしておく必要があります．上記のケースでは，IRQモードはスーパバイザ・モードから1ビットだけクリアすればよいので(表5.1)単純化が可能です．しかし，上のコードで任意の二つの非ユーザ・モード間，または非ユーザ・モードからユーザ・モードへの移動が可能です．

モード変更はMSRの実行後のみ有効になります．結果がCPSRに再コピーされるまで，中間作業はモードに何の変更も与えません．

<<注意>>

(1) ユーザ・モードでCPSR[23:0]を変更しようとしても無駄である．

(2) ユーザ・モードまたはシステム・モードでSPSRにアクセスしようとするのは避ける．これらのモードにはSPSRがないので，予想しない結果になることがあるからです．

5.16 コプロセッサ命令

ARMアーキテクチャは，コプロセッサの追加によって命令セットを拡張する汎用的なメカニズムをサポートしています．もっともありふれたコプロセッサは，ARM720でキャッシュおよびメモリ管理ユニットなどオンチップ機能を制御するシステム・コプロセッサです．浮動小数点ARMコプロセッサも開発されており，特定アプリケーション向けコプロセッサも可能です．

◆コプロセッサ・レジスタ

ARMコプロセッサには自分自身の専用レジスタ・セットがあり，その状態を制御する命令はARMレジスタを制御する命令を反映しています．

フロー制御に責任があるのはARMだけですから，コプロセッサ命令はデータ操作とデータ転送を扱います．RISCロード/ストア・アーキテクチャの原理に基づいて，これらのカテゴリは明確に分かれています．命令形式は，以下を反映しています．

(1) コプロセッサ・データ操作命令

コプロセッサ・データ操作は完全にコプロセッサの内部で実行され，コプロセッサ・レジスタに状態変化を起こします．たとえば，浮動小数点加算ですが，浮動小数点コプロセッサ内の二つのレジスタを加算し，別のレジスタに結果を置くものです．

(2) コプロセッサ・データ転送命令

コプロセッサ・データ転送命令は，コプロセッサ・レジスタの値をメモリへストアしたり，メモリからロードしたりします．各コプロセッサは独自のデータ型をもっている場合があるため，各レジスタに転送するワード数はコプロセッサに依存します．

ARMがアドレスを生成しますが，コプロセッサが転送ワード数を制御します．コプロセッサは転送の一環としてデータ型変換を行うことができます(たとえば，浮動小数点コプロセッサはロードした全値を80ビットの内部表現に変換する)．

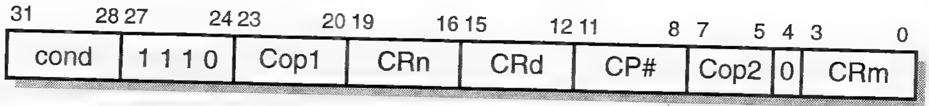
(3) コプロセッサ・レジスタ転送命令

上記に加え，ARMとコプロセッサ・レジスタ間の値の移動が役立つ場合もあります．再び浮動小数点コプロセッサを例にとると，FIX命令で浮動小数点値をコプロセッサ・レジスタから取り，整数に変換，その整数をARMレジスタに移動します．浮動小数点比較は，しばしば制御フローの変更に必要な結果を生成するため，比較結果は必ずARM CPSRに反映されます．

これらすべての命令によって，ARM命令セットの汎用的な拡張が可能となり，特定アプリケーション向けデータ型および機能に対応します．

5.17 コプロセッサ・データ操作命令

コプロセッサ・レジスタ内データに対する内部的処理を制御します．標準的な形式はARM整数データ処理命令の3アドレス形式に従いますが，全コプロセッサ・フィールドについて他の解釈も可能です．



〔図5.15〕コプロセッサ・データ操作命令のバイナリ・エンコーディング

<<概要>>

ARMは，存在するすべてのコプロセッサに対してこの命令を発行します．うち一つでもコプロセッサが受諾すれば，ARMは次の命令に移ります．受諾されなければ，未定義命令トラップを起こします(これは，不在のコプロセッサのソフトウェア・エミュレーションを実現するのに使用できる)．

通常，コプロセッサ番号CP#で認識されるコプロセッサが命令を受諾し，Cop1，Cop2フィールドで定義される処理を実行します．CRn，CRmはソース・オペランドとして用い，結果はCRdに置きます．

<<書式>>

```
CDP{<cond>} <CP#>, <Cop1>, CRd, CRn, CRm{, <Cop2>}
```

<<例>>

```
CDP      p2, 3, Cr0, Cr1, Cr2
CDPEQ    p3, 6, Cr1, Cr5, Cr7, 4
```

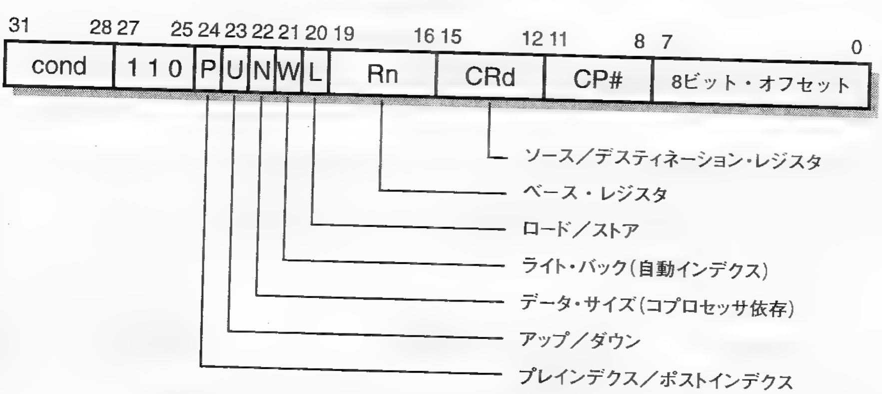
<<注意>>

(1) Cop1，CRn，CRd，Cop2，CRmフィールドの解釈はコプロセッサに依存する．上記の解釈は推薦されるもので，ARM開発ツールとの互換性を最大限にする．

5.18 コプロセッサ・データ転送命令

コプロセッサ・データ転送命令は，前述のワードおよび符号なしバイト・データ転送命令のイミディエート・オフセット形式と似ていますが，オフセットは12でなく8ビットに限られています．

プレインデクス，ポストインデクス・アドレシングをもつ自動インデクス形式も可能です．



〔図5.16〕コプロセッサ・データ転送命令のバイナリ・エンコーディング

<<概要>>

この命令は，存在するすべてのコプロセッサに対して同時要求します．一つも受諾しなければ，未定義命令トラップを発生し，ソフトウェアを使ってコプロセッサのエミュレーションをすることもできます．通常，コプロセッサ番号CP#のコプロセッサがあれば，命令を受諾します．

アドレス計算は，ARM内で行い，ARMベース・レジスタ(Rn)と，8ビット・イミディエート・オフセットを左へ2ビット・シフトしてスケーリングしたワード・オフセットを使います．アドレシング・モードと自動インデクスは，ARMワードおよび符号なしバイト転送命令と同様に制御されます．これにより最初の転送アドレスを定義し，続くワードはインクリメントしたワード・アドレスに転送，もしくはそこから転送します．

データはコプロセッサ・レジスタ(CRd)経由で渡されたり，受け取ったりします．転送するワード数はコプロセッサで制御され，Nビットが二つのうちどちらの長さかを決めます．

<<書式>>

(1) プレインデクス形式：

```
LDC|STC{<cond>}{L} <CP#>, CRd, [Rn, <offset>]{!}
```

(2) ポストインデクス形式：

```
LDC|STC{<cond>}{L} <CP#>, CRd, [Rn], <offset>
```

どちらの場合も，LDCがメモリからコプロセッサ・レジスタへのロードを選択し，STCがコプロセッサ・レジスタからメモリへのストアを選択します．Lフラグがあれば，それは長いデータ・タイプ(N=1)を選択します．<offset>は，#+/-<8-bit immediate>.

<<例>>

```
LDC     p6, Cr0, [r1]
STCEQL  p5, Cr1, [r0], #4
```

<<注意>>

(1) N，CRdフィールドの解釈はコプロセッサに依存する．上記の使用は推薦されるもので，ARM開発ツールとの互換性を最大にする．

(2) アドレスがワード・アラインされていなければ最下位2ビットは無視するが，例外を発するARMシステムもある．

(3) 転送するワード数はコプロセッサが制御し，コプロセッサが転送を終了すると表示するまで(第4章の「データ転送」の項を参照)，ARMはシーケンシャル・アドレスを生成し続ける．データ転送の間，ARMは割り込み要求に応答しない．したがって，コプロセッサ設計者は，長いデータ転送によりシステム割り込み応答時間が損なわれないよう留意しなければならない．

最大転送長を16ワードに制限することにより，コプロセッサ・データ転送が最悪でも複数レジスタ・ロードおよびストア命令を上回らならないようにすることができます．

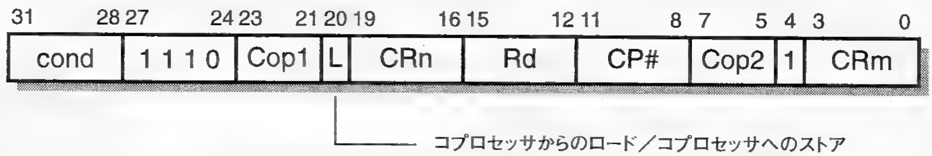
5.19 コプロセッサ・レジスタ転送命令

コプロセッサで生成した整数を，直接ARMレジスタまたはARM条件コード・フラグに転送します．典型的

な用法は以下のとおりです.

(1) ARMレジスタに整数を戻す浮動小数点FIX処理
(2) 浮動小数点比較. 比較結果を直接ARM条件コード・フラグに戻し, 制御フローを決定する.
(3) FLOAT処理. 整数値をARMレジスタから取りコプロセッサに送ると, そこで浮動小数点表現に変換, コプロセッサ・レジスタに置く.

システム制御コプロセッサは, 複雑なARM CPU(中央処理装置)でキャッシュやメモリ管理機能を制御するのに使用するコプロセッサですが, この転送命令によりオンチップ制御レジスタにアクセス, 変更を加えます.



〔図5.17〕コプロセッサ・レジスタ転送命令のバイナリ・エンコーディング

<<概要>>

この命令は, 存在するすべてのコプロセッサに対して同時要求します. 通常, コプロセッサ番号CP#のコプロセッサが命令を受諾します. どのコプロセッサも受諾しなければ, 未定義命令トラップを発生します.

コプロセッサからのロード命令を受諾すると, コプロセッサは通常, Cop1, Cop2が定義する処理をソース・オペランドCRn, CRmに対して実行, 32ビット整数結果をARMに戻し, ARMは結果をCRdに置きます.

コプロセッサへのストア命令を受諾すると, コプロセッサは32ビット整数をARMレジスタRdから受け入れ, 何らかの処理を行います.

コプロセッサからのロード命令の中で, PCがデスティネーション・レジスタRdとして指定されているとき, コプロセッサが生成した32ビット整数のうち最上位4ビットは, CPSRのN, Z, C, Vフラグに置かれます.

<<書式>>

(1) コプロセッサからARMレジスタに移動：

```
MRC{<cond>} <CP#>, <Cop1>, Rd, CRn, CRm{, <Cop2>}
```

(2) ARMレジスタからコプロセッサに移動：

```
MCR{<cond>} <CP#>, <Cop1>, Rd, CRn, CRm{, <Cop2>}
```

<<例>>

```
MCR     p14, 3, r0, Cr1, Cr2
MRCCS   p2, 4, r3, Cr3, Cr4, 6
```

<<注意>>

(1) Cop1, CRn, Cop2, CRmフィールドはコプロセッサが解釈する. 上記の解釈は推薦されるもので, ARM開発ツールとの互換性を最大にする.
(2) コプロセッサが32ビット値をARMに転送する準備として内部作業をしなければならない場合は(たとえば浮動小数点FIX処理では, 浮動小数点値を同等の固定小数点値に変換しなければならない), コプロセ

ッサが転送に取りかかる前に実行しなければならない．したがって，データを準備する間，コプロセッサが「ビジー待ち」状態のハンドシェイクをしなければならないこともよくある．

ARMはビジー待ちの間も割り込みを受けることができ，実際，割り込みを受けるとハンドシェイクから抜け出し，割り込みを処理します．そして多くの場合，割り込み処理ルーチンから復帰後，コプロセッサ命令を再試行しますが，できないこともあります．それは，割り込みによってタスク・スイッチした場合などです．

どちらにしても，コプロセッサは矛盾のない結果を出す必要があります．したがって，ハンドシェイク・コミット以前に実行する準備作業は，コプロセッサの見えている状態を変えてはなりません．

(3) 転送終了後は，コプロセッサで行ういかなるデータ変換作業も可能なので，ARMからコプロセッサへの転送は一般に簡単である．

5.20 ブレークポイント命令（BRK_アーキテクチャversion5Tのみ）

ブレークポイント命令は，ソフトウェア・デバッグを行うときに使用します．この命令により，プロセッサは通常の命令実行を中断し，適切なデバッグ手順を開始するのです．

31 28	27 20	19 16	15 12	11 8	7 4	3 0
1110	00010010	xxxx	xxxx	xxxx	0111	xxxx

〔図5.18〕ブレークポイント命令のバイナリ・エンコーディング

<<概要>>

デバッグ・ハードウェア・ユニットが適切に設定されている場合，この命令によりプロセッサがプリフェッチ・アボートを起こします．

<<書式>>

```
BRK
```

<<例>>

```
BRK    ; !
```

<<注意>>

(1) ARMアーキテクチャversion5Tを実装しているプロセッサだけが，BRK命令をサポートしている（5.23節参照）．

(2) BRK命令は非条件実行するため，条件フィールドは‘ALWAYS‘コードを含む．

5.21 未使用の命令空間

ここまでに，2^{32}の命令ビット・エンコーディングのすべてに意味を割り当てたわけではありません．使用していないエンコーディングは将来の命令セット拡張に使うことができます．

未使用命令エンコーディングは使用済みエンコーディングの余った隙間にあり，将来の用法はその位置を見

れば推測できます.

◆未使用演算命令

5.8節で説明した乗算命令と非常によく似ています．このエンコーディングは，整数除算命令などになる可能性があります.

31 28	27 22	21 20	19 16	15 12	11 8	7 4	3 0
cond	000001	op	Rn	Rd	Rs	1001	Rm

〔図5.19〕演算命令拡張空間

◆未使用制御命令

5.5節で説明した分岐とエクスチェンジ命令，5.14節，5.15節で説明したステータス・レジスタ転送命令を含みます．ここの隙間はプロセッサ動作モードに影響するほかの命令をエンコードするのに使用することもできます.

<table>
<tr><th>31 28</th><th>27 23</th><th>22 21</th><th>20</th><th>19 16</th><th>15 12</th><th>11 8</th><th>7</th><th>6</th><th>4</th><th>3 0</th></tr>
<tr><td>cond</td><td>00010</td><td>op1</td><td>0</td><td>Rn</td><td>Rd</td><td>Rs</td><td colspan="2">op2</td><td>0</td><td>Rm</td></tr>
<tr><td>cond</td><td>00010</td><td>op1</td><td>0</td><td>Rn</td><td>Rd</td><td>Rs</td><td>0</td><td>op2</td><td>1</td><td>Rm</td></tr>
<tr><td>cond</td><td>00110</td><td>op1</td><td>0</td><td>Rn</td><td>Rd</td><td>#rot</td><td colspan="4">8-bit immediate</td></tr>
</table>

〔図5.20〕制御命令拡張空間

◆未使用ロード・ストア命令

5.13節で説明したスワップ命令，5.11節のハーフ・ワードおよび符号付きバイトのロード/ストア命令が占める領域にも未使用エンコーディングがあります．これらは，将来必要性が出た場合，データ転送命令を追加するのによく使用されます.

31 28	27 25	24	23	22	21	20	19 16	15 12	11 8	7	6 5	4	3 0
cond	000	P	U	B	W	L	Rn	Rd	Rs	1	op1	1	Rm

〔図5.21〕データ転送メモリ拡張空間

◆未使用コプロセッサ命令

次の命令形式は，5.18節で述べたコプロセッサ・データ転送命令に似ていて，必要に応じてコプロセッサ命令を追加するのによく使用されます.

31 28	27 24	23 22	21	20	19 16	15 12	11 8	7 0
cond	1100	op	0	X	Rn	CRd	CP#	offset

〔図5.22〕コプロセッサ命令拡張空間

◆未定義命令空間

未定義命令の最大領域は，5.10節で述べたワードおよび符号なしバイト・データ転送命令に似ています．しかし，この空間は完全に自由に使用できます．

31 28	27 25	24 5	4	3 0
cond	0 1 1	XXXXXXXXXXXXXXXXXXXX	1	XXXX

〔図5.23〕未定義命令空間

◆未使用命令の動作

現在のARMプロセッサはすべて，**図5.21**の未定義命令空間に示したエンコーディングにあたる命令を実行しようとすると，未定義命令トラップを起こします．

最新のARMプロセッサは，いかなる未使用の操作コードを実行しても未定義命令トラップを起こしますが，旧いバージョン（ARM6，ARM7を含む）の動作は予測できません．したがって，未使用命令は避けてください．

5.22 メモリ・フォールト

ARMプロセッサでは，メモリ・システム（というよりメモリ管理ユニット）がメモリ・アクセスを拒否することがあります．つまりメモリ・システムがメモリから要求された値を戻さず，メモリ・アクセスが正常に完了しなかったことを表示するのです．そして，プロセッサが例外ハンドラに入り，ソフトウェアが問題から回復しようとします．汎用マシンでもっとも多いメモリ・フォールトは以下のとおりです．

(1) ページ不在

アドレスされたメモリ位置がディスクにページ・アウトされている場合．

仮想メモリ・システムでは，ひんぱんに使われないページはディスクに保持されます．そのようなページにある命令やデータにアクセスしようとすると，MMUがアクセスをアボートするので失敗します．システム・ソフトウェアはアボートの原因を突き止め，求められるページをディスクからメモリにフェッチ，しかるべくMMU内で変換テーブルを変更し，アボートしたアクセスを再試行します．

ディスクからページをフェッチするには時間がかかるため，OSは失敗した処理をスイッチアウトし，転送している間に別のタスクをスケジュールすることもよくあります．

(2) ページ保護

アドレスされたメモリ位置が一時的にアクセス不可になっている場合．

ページがメモリにロードされるとき，OSが最初に読み出しオンリに設定することがあります．ページを書こうとすると失敗し，ページが変更されており，ディスクに再度スワップアウトする場合は保存しなければならないとOSに警告します（変更されていないページは，旧いコピーがあれば，ディスクに再度書き込む必要はない）．

ページング・アルゴリズムの使用に関する統計をとるために，定期的にページをアクセス不可にするOSもあります．

(3) ソフト・メモリ・エラー

メモリにソフト・エラーが検知された場合．

アルファ粒子放射がDRAM記憶セルの状態を変化させるため，大型メモリ・システムはかなりのエラー率があります．メモリ・システムに単純なエラー検知（パリティ・チェックなど）しかない場合，フォールトは回復できないのでフォールトしている処理は終了しなければなりません．メモリ・システムに完全なエラー検出と訂正（ECC）ハードウェアがある場合，プロセッサは通常エラーに気づきませんが，OSがメモリ・エラー率の統計を蓄積できるように，フォールトを発生することもできます．メモリにハードウェア・エラー検知装置はあるがソフトウェアでの誤りに頼っている中間の場合では，誤りは訂正され，再試行のシーケンスがとられます．

◆組み込みシステム

よくある小型組み込みシステムでは，通常，ハード・ディスクは使えず，どのみちディスクへのページングはシステムが満たさなければならないリアルタイム制約に合わないのが普通です．そのうえ，メモリ・システムが小さいので（わずかなメモリ・チップでできた数Mバイト以下のもの），ソフト・エラー率は無視でき，エラー検知が組み込まれていることはまれです．ですから，多くの組み込み型システムはメモリ・フォールトをまったく使用しません．

組み込みシステムにおける一般的用法は，ルーチンのライブラリを圧縮形式にしてROMにストアすることと，仮想メモリ・テクニックを使って各ルーチンへの呼び出しをトラップ，要求どおりRAMへ展開して実行することです．圧縮形式でストアする長所は，サイズとROMコストの低減です．短所は，展開に時間がかかることです．

ほかの用法としては，リアルタイムOSで実行するプロセスに対して多少保護をかけることです．

◆メモリ・フォールト

ARMでは，命令フェッチ（プリフェッチ・アボート）中，データ転送（データ・アボート）中に検知したメモリ・フォールトを別々に処理します．

◆プリフェッチ・アボート

命令フェッチが失敗すると，メモリ・システムがアボート信号を発し（プロセッサへの専用入力），無意味なメモリ・ワードを返します．内部的には，ARMは無意味な命令をアボート・フラグと一緒に命令パイプラインに入れ，そしてメモリがデコード・ステージに入るまで平常どおり動作するのです．メモリがデコード・ステージに入ると，アボート・フラグが命令に優先し，デコーダがプリフェッチ・アボート・ベクタを使って例外エントリ・シーケンスを生成します．

最終的に実行された分岐命令のすぐ後にフェッチされた場合など，アボートした命令が実行されない場合は，例外が起こらずフォールトは無視されます．

◆データ・アボート

データ値を取りにメモリにアクセスしている途中で起こるメモリ・フォールトは，もっと処理が複雑です．メモリ・システムは命令アボートとデータ・アボートを区別する必要がありません．たんに，処理できないアドレスを見つけて入力アボートを発するだけです．しかし，プロセッサはデータ・アボートに対応するほうが多く動作しなければなりません．なぜなら，データ・アボートは実行中の命令に関する問題であり，プリフェッチ・アボートはデコーダにまだ入っていない命令に関する問題だからです．

場合によっては，フォールトの原因後に命令を再試行するのが目的なのですから，命令の状態（つまりレジ

スタ値）が命令アボート後と実行前とで変わっていないようにしなければなりません．もしくは少なくとも，状態が十分回復でき，命令を再度実行した後の状態が，最初に実行できればそうなったであろう状態と同じになるようにしなければなりません．

◆LDMデータ・アボート

これがどんなに難しいかを知るために，r0をベース・レジスタとするレジスタ・リストに16本のレジスタがある複数ロード命令を考えましょう．最初，アドレスには問題ありませんから，ロードを開始します．最初に来たデータ値がr0を上書きし，ほかのレジスタも最後のアドレス（デスティネーションはPC）がページ境界を越えて失敗するまで，続けて書き込まれていきます．プロセッサ状態はほとんど失われています．どうすれば回復できるでしょうか？

アボート信号は，PCが上書きされる直前に出ますから，少なくともフォールトを起こした命令のアドレスだけは残っていることになります．しかし，ベース・レジスタはとうの昔に失っていますから，命令の再試行はどうしたらいいのでしょう？　幸運にも，命令の進行中，プロセッサはベース・レジスタ値（自動インデクス後のこともあります）のコピーを片隅に残しています．したがって，本来，PCアクセスが失敗していなければ，命令の最後はPCを新しい値に変更することでしたが，その代わりこの保存してあった値をベース・レジスタに再コピーするのです．

残っていたPC，そして（変更された）ベース・レジスタがあり，ほかのレジスタが上書きされています．ベース・レジスタについては，命令を調べてリスト内のレジスタ数とアドレシング・モードを決定すれば，ソフトウェアで逆行可能です．上書きしたレジスタは命令再試行の際，もう一度正しい値をロードするところですから，これでいいのです！

◆歴史に関して

データ・アボートから回復する条件は，最初のARMプロセッサ・チップの開発中，最後のほうで加えられました．それまで，基底レジスタを始点としてモードにしたがい，メモリのインクリメント，デクリメントを重ね，さまざまな複数ロードおよびストア・アドレシング・モードが実現されていました．したがってチップは，アドレス・インクリメンタ/デクリメンタ・ユニットで設計されていました．

仮想メモリをサポートする必要がはっきりしたとき，デクリメント・モードがアボート回復を大幅に困難にしていることがわかりました．なぜなら，PCがアボート信号の前に上書きされる可能性があるからです．したがって，アドレスはつねにインクリメントするよう実装に変更が加えられました．使用するメモリ・アドレスは同じ，レジスタからメモリへのマッピングも同じ（図3.2参照），ただ，転送順序だけ最下位アドレスを最初，PCを最後にするよう変更したのです．

この変更は実行時期が遅かったので，アドレス生成論理のレイアウトは変えられませんでした．そのため，初代ARMシリコンのアドレス・インクリメンタ/デクリメンタは常にインクリメントするようにハードワイヤドされているのです．この冗長性はその後の実現に向けては受け継がれなかったことはいうまでもありません．

◆アボートのタイミング

プロセッサがメモリ・システムからフォールト表示を受けるのが早ければ早いほど，状態が保存しやすくなります．プロセッサがフォールト信号を要求するのが早ければ早いほど，メモリ・システム設計は難しくなります．したがって，プロセッサのフォールト処理がアーキテクチャ的に単純であることと，メモリ・システムの工学的効率性とを両立させるには，微妙な問題です．

この微妙さは，メモリ管理ユニットだけに影響するのではありません．プロセッサにキャッシュ・メモリがあれば，キャッシュ設計もアボート・タイミングによって変わってきます．

初期のARMプロセッサでは，アボート信号はクロック・サイクルのフェーズ1の終わり，メモリ・アクセスの半分以前に要求します（図4.8参照）．CAM-RAM（CAMはContent Addressable Memory：内容アドレス可能メモリ）をもつフルアソシアティブ・キャッシュ機構は，フェーズ1でアクセスが順調なことを確認するかプロセッサを止めることができるよう，CAMアクセスだけからヒット・ミス信号を生成するよう設計することができます．プロセッサさえ止まれば，MMUはアボート信号の生成を制御する時間があります．一方，セット・アソシアティブ・キャッシュの場合，通常はサイクルの終わりでヒット・ミス信号を生成するため，（アボートを生成する）MMUまでミスを延ばすには遅すぎ，性能が大きく下がることになります（ARM710にはセット・アソシアティブRAM-RAMキャッシュがあるため上の規則にはしたがわない．フェーズ1の終わりまでに，キャッシュはヒット・ミス信号，MMUは保護情報を生成する）．

キャッシュとMMU設計にかかる制限を和らげるため，新しいARMは，読み取ったデータへと同様のタイミングで，サイクルの終わりでアボートもフラグ付けされるよう設計し直されました．ただし，プロセッサ状態が変化しているためアボート回復ソフトウェアはより多くの作業をしなければなりません．

前後どちらのアボート・タイミングにも対応するよう設定（外部ハード・ワイヤリング，またはLビット，CP15レジスタ1のビット6などの使用については，11.2節を参照）できるARMプロセッサもあります．

◆ARMのデータ・アボート

データ・アボート後のARMの状態はプロセッサによって違い，プロセッサによってはアボート設定が前後どちらかによっても違います．

(1) いかなる場合でもPCが保存されます（データ・アボート例外において，エントリr14_abtにフォールトしている命令＋8バイトのアドレスがある）．

(2) ベース・レジスタは変更されていないか，または自動インデクスによって変更された値をもちます（ロードした値では上書きされない）．

(3) ほかのロードのデスティネーション・レジスタは上書きされているかもしれませんが，命令再試行の際に正しい値がロードされます．

なぜなら，ベース・レジスタは自動インデクスによって変更される可能性がありますので，いくつか（あまり使われないもの）の自動インデクス・モードは避けてください．例は次のとおりです．

```
        LDR     r0, [r1], r1
```

この命令は，r1をロードのアドレスとして用い，ポストインデクスを使ってr1を自分に加えます．その処理中にトップ・ビットを失います．データ・アボートの後，修正済みのr1値しかない場合は，元の転送アドレスを回復することは不可能です．一般に，一つのアドレシング・モードで，同じレジスタをベースとインデクス両方に使うことは避けるべきです．

5.23 ARMアーキテクチャのバリエーション

ARMアーキテクチャは開発の経過で改良を重ねてきました．以下，多様なアーキテクチャ・バージョンについて説明します．

◆バージョン1

ARMアーキテクチャ・バージョン1とは，Acorn Computers Limitedが1983年から85年までに開発した初代ARMプロセッサを指します．初代ARMチップは，26ビット・アドレシング(下記参照)だけに対応し，乗算，コプロセッサはサポートしていませんでした．製品としては，BBCマイクロコンピュータのARMセカンド・プロセッサ・アタッチメント内に使用しただけです．数もわずかでしたが，商用的利用された最初の1チップRISCマイクロプロセッサとしてARMを確立しました．Acorn内部ではArchimedesパーソナルワークステーションのプロトタイプに使用されました．

◆バージョン2

ARM2チップは，Acorn ArchimedesとA3000製品に搭載され大量に販売されました．依然として26ビット・アドレス機でしたが，32ビット結果乗算命令とコプロセッサ・サポートが付いていました．ARM2は，現在ARM LimitedがARMアーキテクチャ・バージョン2と呼ぶアーキテクチャを採用しています．

◆バージョン2a

ARM3チップは，オンチップ・キャッシュをもつ最初のARMでした．アーキテクチャはバージョン2と非常に似ていましたが，不可分なロードおよびストア(`SWP`)命令を加え，キャッシュ管理のためにコプロセッサ15をシステム制御コプロセッサとして導入しました．

◆バージョン3

1990年にARM Limitedが独立後，最初に設計したARMプロセッサはARM6でした．ARM6はマクロセル，スタンドアローン・プロセッサ(ARM60)として，またオンチップ・キャッシュ，MMU，ライト・バッファ(Apple Newtonで使用されているARM600，ARM610)付き統合CPUとして販売されました．ARM6はARMアーキテクチャ・バージョン3を導入しましたが，バージョン3は32ビット・アドレシング，別々のCPSR，SPSRをもち，コプロセッサ・エミュレーション，スーパバイザ・モードでの仮想メモリ・サポートが可能な未定義およびアボート・モードを追加していました．

ARMアーキテクチャ・バージョン3はバージョン2aと後方互換性があり，ハードワイヤド26ビット処理，処理ごとの26，32ビット混在処理，どちらも可能です．

◆バージョン3G

ARMアーキテクチャ・バージョン3Gは，バージョン2aとの後方互換性のないバージョン3です．

◆バージョン3M

ARMアーキテクチャ・バージョン3Mは，フル64ビット結果を生成する符号付きおよび符号なし乗算および積和命令を導入しました．

◆バージョン4

ARMアーキテクチャ・バージョン4は，符号付きおよび符号なしハーフ・ワード，符号付きバイト・ロードおよびストア命令を加え，`SWI`空間の一部をアーキテクチャ定義の処理にあてています．システム・モード(ユーザ・レジスタを使う特権モード)が導入され，命令空間のいくつかの未使用部分が未定義命令としてトラップされます．

この段階で，初期のARMにおいて「pc+12」を生成していたr15の用法が予想外の結果を出す恐れがあることがはっきりしました(したがって，バージョン4準拠の実装は「pc+12」動作を再生する必要がない)．これは，完全なフォーマット定義をもつ最初のバージョンでした．

◆バージョン4T

ARMアーキテクチャ・バージョン4Tは，16ビットThumb圧縮形式命令セットを導入しました．

◆バージョン5T

ARMアーキテクチャ・バージョン5Tは，最近発表されたものであり，本書執筆中の現在は，ARM10プロセッサでのみサポートされています(ARM10はまもなくversion5TEもサポートする)．これはバージョン4TにBLX，CLZ，BRKを加えたスーパーセットです．

◆バージョン5TE

バージョン5TEは，アーキテクチャ5Tに8.9節で説明する信号処理命令セット拡張を加えています．

◆まとめ

表5.7は，各コアが使用するARMアーキテクチャ・バージョンをまとめたものです．

〔表5.7〕ARMアーキテクチャーのまとめ

コア	アーキテクチャ
ARM1	v1
ARM2	v2
ARM2aS, ARM3	v2a
ARM6, ARM600, ARM610	v3
ARM7, ARM700, ARM710	v3
ARM7TDMI, ARM710T, ARM720T, ARM740T	v4T
StrongARM, ARM8, ARM810	v4
ARM9TDMI, ARM920T, ARM940T	v4T
ARM9E-S	v5TE
ARM10TDMI, ARM1020E	v5TE

5.24 例題と演習

(3.4節，3.5節も参照してください．)

＜例題5.1＞ スカラ積プログラムで5.8節にあるものよりも高速に処理するプログラムを書きなさい．

元のプログラムは10サイクルと，各データ値ペアにつき乗算時間(データによる)がかかります．各データ値のロードには3サイクル，分岐はループごとに3サイクルかかります．

ロードにかかるサイクルは複数レジスタ・ロードで低減でき，分岐については「ループ・アンローリング」で低減できます．2方法を組み合わせることにより，次のようなプログラムができます．

```
            MOV     r11, #20            ; initialize loop counter
            MOV     r10, #0             ; initialize total
LOOP        LDMIA   r8!, {r0-r3}        ; load 4 1st vector values..
            LDMIA   r9!, {r4-r7}        ; ..and 4 2nd vector values
            MLA     r10, r0, r4, r10    ; accumulate 1st product
            MLA     r10, r1, r5, r10    ; accumulate 2nd product
            MLA     r10, r2, r6, r10    ; accumulate 3rd product
            MLA     r10, r3, r7, r10    ; accumulate 4th product
            SUBS    r11, r11, #4        ; decrement loop count by 4
            BNE     LOOP                ; loop if not finished
```

この場合，ループ・オーバヘッドは16サイクルとデータ値4ペアの乗算時間，つまり各ペアにつき4サイクルです．

演習5.1.1 あるメモリ位置から別のメモリ位置へバイト文字列をコピーするサブプログラムを書きなさい．ソース・ストリングの始点はr1で，長さ（バイト数）はr2で，デスティネーション・ストリングの始点はr3でパスします．

演習5.1.2 上の例で出てきたテクニックを用いて性能を向上させながら，再度上の演習を繰り返しなさい．ソース・ストリングとデスティネーション・ストリングの両方ともワード・アラインしてあり，ストリングの長さは16バイトの倍数とします．

演習5.1.3 ソース・ストリングはワード・アラインしてありますが，デスティネーション・ストリングは任意のバイト・アラインとします．ストリングの長さは，やはり16バイトの倍数です．ブロックのストアには複数レジスタ転送を使い，エンド条件にはバイト・ストアを使いながら，16バイト・ブロックを一度に処理するプログラムを書きなさい．

第6章 高級言語のための アーキテクチャでのサポート

高級言語を使えば，データ型，構造体，手続き，関数などの抽象化によるプログラミングができます．RISCアプローチは，高級な概念を直接サポートする命令セットをもちませんが，原始的なRISC命令セットを使っても高級言語に必要なサポートは十分に可能です．

本章では，高級言語がアーキテクチャに課す要件，またそれをどのように満たすかを考えます．高級言語としてC(Cを例にとるべきではないという意見もあるだろうが)，言語をコンパイルするアーキテクチャとしてARM命令セットを例にとります．

分析の過程で明らかになることですが，ARMのようなRISCアーキテクチャではコンパイラの書き手にかなりの裁量が要求されます．裁量しだいでは，異なるソース言語から生成したルーチンでプログラムを簡単に構築することもできます．これは大切な問題ですから，書き手がエントリとエグジット条件の整合性をもたせるための「ARMプロシージャ・コール標準」も定義されています．

コンパイラ間の合意により有利になる分野として，浮動小数点処理があります．ここでは，ARMハードウェア命令セットに定義されていないデータ型を使用します．

6.1 ソフトウェア設計における抽象化

ソフトウェア設計に重要な抽象化レベルについては前の章で触れました．第5章で述べたようにARMプロセッサのエッセンスはARM命令セットにあります．第9章でARMアーキテクチャの実装にはさまざまな方法があることを説明しますが，肝心なことはプログラマが実装の詳細まで心配しなくてもいいようにすることです．プログラムがある実装で正しく動作すれば，ほかのすべての実装でうまく動作するはずです(一定の条件はあります)．

◆アセンブリ・レベルの抽象化

アセンブリ・プログラミング・レベルで書くプログラマは，命令，アドレス，レジスタ，バイト，ワードなどを使いますが，(ほとんど)直接，生の機械語レベル命令セットを扱うことになります．

優秀なプログラマは，複雑な課題を抱えた場合，プログラム設計を単純化するために，高レベルの抽象化を求めます．たとえば，グラフィック・プログラムでは，線を引くことが多くあるので，始点と終点の座標を指定すれば線を引くようなサブルーチンが便利になります．あとは，プログラムは座標のみ扱えばいいのです．

アセンブリ・プログラミング・レベルで抽象化は大切ですが，抽象化のサポートと機械語要素を使っての表現はプログラマに全責任がかかっています．したがってプログラマは機械語要素をよく理解しなければなりませんし，しばしば機械語レベルに戻って考える心構えが必要です．

◆高級言語

高級言語を使えば，プログラマが機械語レベルより上の抽象化で考えることができます．実際，どのマシンの上でプログラムを最終的に動作するのか知らなくてもいいのです．レジスタ数などパラメータはアーキテクチャによって異なりますから，当然，言語設計に反映されてはいけません．

ターゲット・アーキテクチャにおいて高級言語で使用する抽象化をサポートするのが「コンパイラ」です．コンパイラ自身も非常に複雑なソフトウェアで，コンパイラが生成するコードの効率性は，ターゲット・アーキテクチャがどれだけコンパイラをサポートするかによって大幅に差が出ます．

かつて，コンパイラをサポートする最良の方法は命令セットをより複雑にし，言語の高級処理を直接実装することだと信じられていました．RISC思想の導入で流れは逆転し，命令セット設計では柔軟な原始的処理に焦点を絞り，コンパイラがそこから高級処理を構築するようになりました．

本章では，高級言語の条件と，その条件をRISC思想に基づいたARMアーキテクチャで満たす方法を述べます．

6.2 データ型

便利とはいえませんが，どんなコンピュータ・プログラムでもブール論理変数(以下，ブール変数)の「真'1'」と「偽'0'」で表現することができます．なぜなら，ゲート・レベルのハードウェアが処理できるのは'1'と'0'だけだからです．

ARM命令セットの定義では，抽象化を導入しており，プロセッサの機能を命令，バイト，ワード，アドレスなどで表現します．各用語は，一定の観点から見た論理変数の集まりを示しています．たとえば，一つの命令，データ・ワード，アドレスはすべて32ビット長で，そのうち一つを含む32ビット位置は違うデータ型のものを含む位置と区別できません．違いは，情報のストア方法にあるのではなく，情報の用途にあります．したがってコンピュータのデータ型は，次の点で特徴づけられます．

- 要求するビット数
- ビットの順序
- ビット・グループの用途

アドレスや命令のようなデータ型は，主にコンピュータの目的を果たすために存在しますが，ほかは人間のユーザがアクセスできる形で情報を表現するために存在します．計算のコンテクストにおいて，後者のカテゴリのもっとも基本的なものは数です．

◆数

そもそも夜中に羊が盗まれていないかどうかを確かめる手段として発達したのであろう単純な概念が，何百年を経て複雑なメカニズムに進化し，1mmの何千分の1の大きさで，1秒に何億回もスイッチするトランジスタの動作を計算するようになりました．

(1) ローマ数字

これは人間が書いた数字です．

```
MCMXCV
```

(2) 10進法

上記のローマ数字の解釈は複雑です．シンボルの値はシンボルと位置関係によって決まります．この表記は

大部分(完全にではありません．本の最初の部分を見てください)「10進法」に変わりました．前と同じ数字は，次のようになります．

```
1995
```

右端の桁が1の位，そして左の桁へ順に10，100，1000の桁となっていきます．左へ一つ動くたび，桁の値は10倍されます．

(3) BCD(binary code decimal：2進化10進)

10進数をブール変数群で表すもっとも簡単な方法は，まず10進の各桁の表記を考え，次に各桁の数を四つのブール変数を使って表現することです．0から9までの異なる数をブール変数で表現するには，四つのブール変数が必要です．したがって，論理ゲートで処理できる第1の形式は次のとおりです．

```
0001 1001 1001 0101
```

これは10進法のバイナリ・エンコーディングで，一部のコンピュータがサポートしていて，ポケット計算機ではよく使われています．

(4) バイナリ表記

多くのコンピュータはほとんどの場合，人間主体の10進法をすっかり廃止して純粋な2進法表記を使っています．同じ数は次のように表されます．

```
11111001011
```

ここで右端の桁上げは1の位，次は2の位，そして4の位のようになります．左へ動くたびに桁の値は2倍になります．ある桁における2の値は左隣の桁の1で表現されますから，どの桁も0か1の値で足ります．したがって，2進法の各桁(ビット)は単純なブール変数で表現できます．

(5) 16進法表記

機械は内部的には広く2進法を使っているものの，32ビットのバイナリ数は覚えにくいものです．しかし，見慣れた10進法に変換する(手間が大きく，エラーも多くなる)代わりに，コンピュータの世界では「16進法」(基数16)で表記します．バイナリ数は四つの桁を1組として分割でき，各組は16進法で置き換えられますから簡単です．基数16の場合，0から15のシンボルが必要なため，アルファベットの最初の文字が10進法のシンボルで足りない分を補います．つまり0から9をそのまま用い，10から15はAからFで表します．例の数字は次のようになります．

```
7CB
```

(かつて，16進法を使うのと同じ目的で基数8の8進法が一般的だったこともあった．この場合，アルファベットを使う必要はなくなるが，3ビット1組より4ビット1組のほうが処理しやすかったため，8進法はほとんど使われなくなった．)

◆数の範囲

紙の上に書くとき，私たちは10進法の桁数を必要なだけ使って書きたい数字を表現します．コンピュータは通常，一つの数字用にあてるビット数を固定してあるので，大きすぎる数値は表記できません．ARMは32ビットは効率よく扱えるので，アーキテクチャがサポートする第1のデータ型は32ビット(符号なし)整数で，次の範囲をもちます．

$$0 \sim 4{,}294{,}976{,}295_{10} = 0 \sim \mathrm{FFFFFFFF}_{16}$$

(下付き数字は基数を表すので，上の範囲は最初10進法，次に16進法で表されている．16進法の値「F」がすべて「1」のバイナリ数を表すことに注目してほしい．)

これで十分な範囲に見えますし，実際，ほとんどの目的には足りますが，プログラマはその限界を認識しなければなりません．というのは，たとえば範囲内での最高値の近くの符号なし数を二つ加えれば間違った答えが出るからです．これは，正しい答えが32ビット内で表現できないからです．プログラム状態レジスタのCフラグが，唯一，何かが間違っていて答えが信用できないことを示します．

大きな数が小さな数から減算されると結果は負になり，いかなる大きさの符号なし整数を使っても表せません．

◆符号付き整数

たいていの場合，正だけでなく負の数も表現できるほうが便利です．そこでARMは，最上位ビットの値が負である2の補数のバイナリ表記もサポートします．つまり32ビットの符号なし整数では，ビット31以外，全ビットが符号付きの場合と同じ値をもちます．ビット31は$+2^{31}$でなく，-2^{31}の値をもちます．ここで，数の範囲は次のとおりです．

$$-2{,}147{,}488{,}148_{10} \sim +2{,}147{,}488{,}147_{10} = 80000000_{16} \sim \mathrm{7FFFFFFF}_{16}$$

数の符号は，ビット31のみで決まり，正の整数値は符号なし整数の値とまったく同じ表現になります．

多くのプロセッサと同じく，ARMは符号付き整数に2の補数表記を使います．なぜなら，それらの加算，減算には，符号なし整数の場合とまったく同じブール論理関数が使用できるからです．したがって，別々の命令をもつ必要はありません(例外はフル64ビットの乗算．ここでARMは別々の符号付きおよび符合なし命令をもつ)．

符号付き整数をサポートするためのアーキテクチャ上の仕組みは，プログラム・ステータス・レジスタのVフラグです．これは，オペランドが符号なしの場合は不要ですが，符号付きオペランドが混じると**オーバフロー**(範囲外)エラーを示すものです．ソース・オペランドは32ビット値で表現されているため範囲外はありえませんが，範囲の限界近くにある2数を加算もしくは減算すると，結果は範囲外に出ます．その32ビット表現は範囲内数ですが，値も符号も間違っています．

◆ほかの数のサイズ

ARMでの標準的な数表現は，符号付きまたは符号なし32ビット整数です．実際，プロセッサ内部ではそれに対応した構造をもちます．ただし，すべてのARMプロセッサは，符号なし8ビット(バイト)ロードおよびストアを実行するので，32ビット・ワードよりもメモリ中で小さい空間を占める小さい正の数を許します．ごく初期のプロセッサ以外は，小さな値をストアするのに必要なメモリを減らすために，符号付きバイト，符号付きおよび符号なし16ビット転送もサポートしています．

32ビット整数が小さすぎれば，大きな数字は複数ワードおよび複数レジスタを使って処理できます．ステータス・レジスタのCフラグを使って下位のワードから上位のワードへキャリ(桁上げ)を伝播すれば，64ビット加算は二つの32ビット加算で実行できます．

```
; 64-bit addition of [r1,r0] to [r3,r2]
ADDS    r2, r2, r0        ; add low, save carry
ADC     r3, r3, r1        ; add high with carry
```

◆実数

ここまで整数だけを考えてきました．実数(real numbers)とは，物理的量を扱うのに便利な分数や超自然数を表すのに使われます．

コンピュータにおける実数の表現は重要な問題なので，次節で取り上げます．ARMコアは，実数データ型をサポートしていませんが，ARM Limitedはそれを処理するタイプや命令群を定義しています．これらの命令は浮動小数点コプロセッサで実行するか，ソフトウェアでエミュレートします．

◆プリント可能文字

数字の次にもっとも基本的なデータ型はプリント可能文字です．標準的なプリンタを制御するためには，上付き，下付き文字，0から9までの10進数字，句読点，特殊文字£，$，%などすべての通常文字を表現する方法が必要です．

◆ASCII

これらすべての異なる英数文字を数えると，すぐに100近くになります．特殊文字などのバイナリ表現は何年か前に7ビットASCII(American Standard for Computer Information Interchange)コードとして規格化されました．その中には，これらのプリント可能文字，「キャリッジ・リターン」，「ライン・フィード」，「ベル」などテレタイプ時代に由来する名前をもつ制御コードがあります．

ASCII文字をコンピュータにストアする通常の方法は，8ビット・バイトに7ビット・バイナリ・コードを入れることです．多くのシステムはコードの用法を拡張しており，たとえば，バイト内の他の128バイナリ数が特殊文字(アクセント付き文字など)を表現する8ビットISO文字セットなどがあります．もっとも柔軟な文字表現は16ビット「Unicode」で，そのような8ビット文字セットを一つのエンコーディングの中に数多く取り入れています．

8ビットのプリント可能文字としてエンコードすると「1995」は次のようになります．

$$00110001\ 00111001\ 00111001\ 00110101 = 31\ 39\ 39\ 35_{16}$$

◆ARMの文字サポート

ARMアーキテクチャにおける文字処理サポートは，符号なしバイト・ロードおよびストア命令です．これは小さな符号なし整数をサポートできると述べましたが，むしろASCII文字を転送するのにひんぱんに使用されます．ASCIIに定義された特定のコードを反映しているものは，ARMアーキテクチャにありません．いかなるエンコーディングでも8ビット以下なら同様にサポートしています．しかし，大した理由もなく異なる文字コードを選ぶのは，最近では正道を踏み外すことになります．

◆バイト順序

例にあげたASCIIは，潜在的な問題を露呈しています．つまり，左から右へ読むように書かれていますが，32ビット・ワードとして読まれるなら，最下位バイトは右端にあるからです．文字出力ルーチンは，文字を増加していく連続したバイト・アドレスとしてプリントするかもしれません．そうなると「リトル・エンディアン」アドレシングにより，「5991」となってしまいます．

メモリ内でワードのバイト順序については，注意しなければなりません(ARMはリトル・エンディアン，ビッグ・エンディアンそれぞれのアドレシングで動作可能である．第5章「メモリ構成」を参照のこと)．

◆高級言語

高級言語はその仕様で必要なデータ型を定義しています．通常，動作するアーキテクチャについては言及していません．しかし，データ型を表現するビット数はマシンがもっとも効率的なサイズを使うため，アーキテクチャによって違うこともあります．

◆ANSI C基本データ型

American National Standards Institute (ANSI) が定義したC言語の一方言ともいえるのが「ANSI標準C」または単純に「ANSI C」と呼ばれるものです．これは次の基本データ型を定義しています．

- 最低8ビットの符号付き，符号なし**文字**
- 最低16ビットの符号付き，符号なし**短整数**
- 最低8ビットの符号付き，符号なし**整数**
- 最低32ビットの符号付き，符号なし**長整数**
- **浮動小数点数，倍精度浮動小数点数**および**拡張倍精度**浮動小数点数
- **列挙型**
- **ビット・フィールド**

ARMのCコンパイラは，これらの型の最小サイズを採用していますが，標準の整数は32ビットを使っています．なぜなら，32ビットがもっともひんぱんに使用されるデータ型であり，ARMは16ビット処理より32ビット処理のほうが効率的だからです．

列挙型は（指定された可能な値の中から一つの値を変数がもつ），必要な値域をもつ最小の整数タイプとして実装されています．ビット・フィールド型（ブール変数群）は整数内に実装されます．複数のビット・フィールドが一つの整数を共有することもあります．その際，最初に宣言されたものが最下位ビット位置をもつと宣言しますが，ワード境界をまたぐことはありません．

◆ANSI C派生データ型

さらに，ANSI C標準は次の派生データ型を定義しています．

- 同じ型の複数オブジェクトの**配列**
- 与えられた型のオブジェクトを戻す**関数**
- 様々な型のオブジェクトの並びを格納する**構造体**
- 与えられた型のオブジェクトを指す**ポインタ**（通常はマシン・アドレス）
- 異なるタイプのオブジェクトが同じ空間を占めることを可能にする**ユニオン**

ARMのポインタは32ビット長（ARMネイティブ・アドレスのサイズ）で，符号なし整数に似ていますが，異なる演算規則に従うことに注意してください．

ARM Cコンパイラはバイト境界で文字をアラインし（つまり各アドレスごと），短整数は偶数アドレスで，ほかのタイプはワード境界でアラインします．構造体はつねにワード境界からはじまり，最初のフィールドから順に上述の整列規則にしたがって可能なかぎり詰め込むようにフィールドを配置します（「パック」構造はアラインメント規則を破っている．メモリの使用については6.9節で詳しく述べる）．

◆Cデータ型に対するARMアーキテクチャ・サポート

上のとおり，ARM整数コアには，符号付きおよび符号なし32ビット整数，符号なしバイトのネイティブサポートがあり，Cの整数（これらを32ビット値として実装するという仮定をしている），長整数，符号なし文字

タイプもカバーします．ポインタはARMのネイティブなアドレスとして実装されているので，直接サポートされています．

ARMアドレシング・モードは，配列と構造体を合理的にサポートしています．ベース＋スケールしたインデクス・アドレシングにより，2^nバイト・サイズのオブジェクト配列のスキャンが可能です．その際，ポインタは配列の始点を指しループ変数はインデクスとして使います．ベース＋イミディエート・オフセット・アドレシングは構造体内のオブジェクトへのアクセスを可能にします．しかし，さらに複雑なアクセスについては追加アドレス計算命令が必要です．

現行のARMバージョンは，符号付きバイト，符号付きおよび符号なし16ビット・ロードとストアを含んでおり，短整数と符号付き文字型を一部ネイティブ・サポートしています．

浮動小数点型については次のセクションで述べますが，基本のARMコアは，ほとんどこれらの型を直接サポートしていないことに注意してください．この型（と，それを操作する命令）は，特定の浮動小数点サポート・ハードウェアがない場合は，複雑なソフトウェア・エミュレーション・ルーチンで処理します．

6.3 浮動小数点データ型

浮動小数点数は実数を一様の精度表現する試みです．この形式で実数を表現する一般的な方法は次のとおりです．

$$R = a \times b^n \qquad \text{（式13）}$$

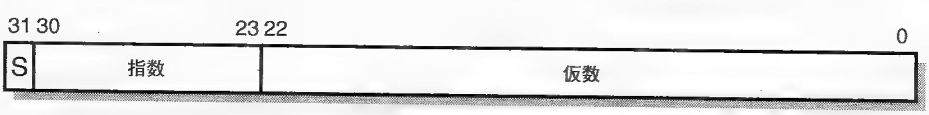
nはaが定義された値域内になるよう選択されています．bは通常データ型のなかで暗黙に指定され，多くの場合は‘2’となります．

◆IEEE754

コンピュータで浮動小数点数を処理するのに，異なるマシンで同じプログラムを動作した時に矛盾のない結果が出るようにするには，複雑な問題が多くあります．1985年にIEEEバイナリ浮動小数点演算標準（ANSI/IEEE標準754-1985，単にIEEE754と呼ばれることもある）が導入されてから，この矛盾性の問題はかなり軽減されました．これは，浮動小数点数の表現方法，実行される計算の精度，エラーの検知と復帰方法などを詳しく定義しています．

IEEE754に定義された浮動小数点数のもっとも簡単な表現は，32ビット「単精度」形式です．

◆単精度



〔図6.1〕IEEE754単精度浮動小数点形式

数は符号ビット（‘S’）と，+127の「バイアス」をもつ（正規化した数の場合）符号なし整数値の指数部，仮数部から成り立っています．ここには見慣れない用語がたくさんあるかもしれません．それを説明するために，数をどう認識するかを考えてみましょう．「1995」をこの形式に変換してみます．

まず，前にも紹介した1995のバイナリ表現を見てみましょう．

```
11111001011
```

これは正の数ですからSビットはゼロになります．

◆正規化数

まず，数を「正規化」します．つまり，（式13）にある形式に数を変換し，$1 \leqq a < 2$で$b = 2$にします．その数のバイナリ形式を見てみると，最初の‘1’の後ろに小数点を挿入することによって，aをこの範囲に制限することができます．バイナリ整数表現におけるバイナリ・ポイントの明示的位置は，右端の桁の右です．したがって，これを左へ10桁動かします．すると，1995の正規化表現は次のようになります．

$$1995 = 1.1111001011 \times 2^{1010} \qquad \text{（式14）}$$

ここで，aとnはどちらもバイナリ表記です．

どのような数を正規化するときも，aにおける小数点の前のビットは‘1’になります（さもなければ，数は正規化されているとはいえない）．したがって，このビットをストアする必要はありません．

◆指数部のバイアス

最後に，この形式は非常に大きい数字だけでなく小さい数字を表現するにも必要ですから，正規化したときに負の指数部を要求する数もあります．符号付き指数部を使う代わりに，標準では「バイアス」を指定しています．バイアス（単精度正規化数の場合は+127）は，指数部に加えます．したがって1995は次のように表現されます．



〔図6.2〕「1995」のIEEE754単精度表現

指数部は，127+10=137で，仮数部は23ビット・フィールドを埋めるために右へゼロ拡張されています．

◆正規化値

一般に，32ビット正規化数の値は次のように求めます．

$$\text{値（正規化数）} = (-1)^{S} \times 1.\text{仮数} \times 2^{(\text{指数}-127)} \qquad \text{（式15）}$$

このフォーマットは広範囲の値を効率的に表現しますが，問題が一つあります．つまりゼロを表現する方法がないことです．したがってIEEE754規格では，指数部がゼロまたは255である数は特別な値を表現するものとして扱います．

- ゼロはゼロの指数部および仮数部で表現します（どちらも符号付きの値なので，正負それぞれのゼロが表現できる）．
- プラスまたはマイナス無限大は，最大指数値とゼロの仮数部と，適切な符号付きビットによって表現します．
- NaN（**非数**：Not a Number）は，最大指数値と非ゼロ仮数で表示します．「静かな」NaNは最上位仮数ビット位置に‘1’をもち，「信号付き」NaNはその位置に‘0’をもちます（ただし，ほかの位置に‘1’がなければ

ならない．さもなければ，無限大に見える）．

- **非正規化**（denormal）数とは，この形式内で正規化するには小さすぎる数で，ゼロの指数部，非ゼロの仮数部，そして次の方程式によって得られる値をもちます．

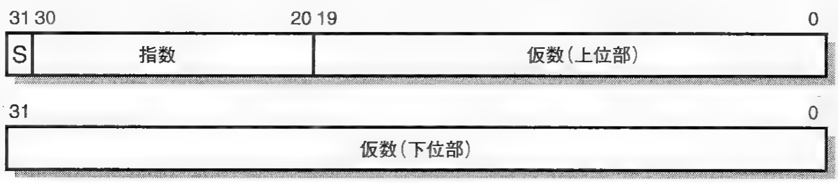
$$値(非正規化数) = (-1)^{S} \times 0.仮数 \times 2^{(-126)} \qquad (式16)$$

NaN形式は，負の数の対数をとるなど無効な浮動小数点処理の結果を表現するのに使います．そして，一連の演算において中間エラー状態がチェックされなかった場合，一見すると無効そうに見える結果を生成するのを防ぎます．

◆倍精度

多くの用途で，単精度形式の精度は不十分です．さらに精度を上げるには，64ビットを使って各浮動小数点値をストアする倍精度形式を使います．

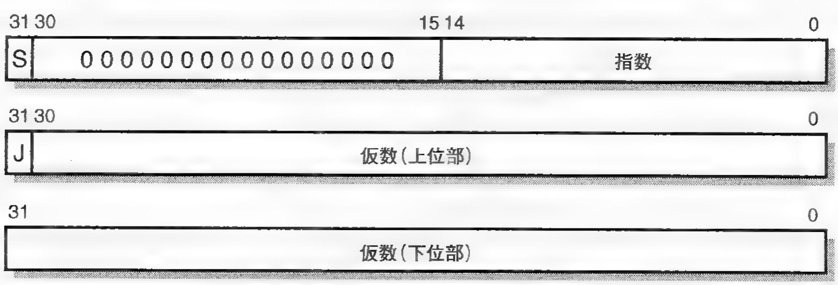
解釈は単精度のものと似ていますが，正規化数の指数バイアスは+1023です．



〔図6.3〕IEEE754倍精度の浮動小数点数形式

◆拡張倍精度

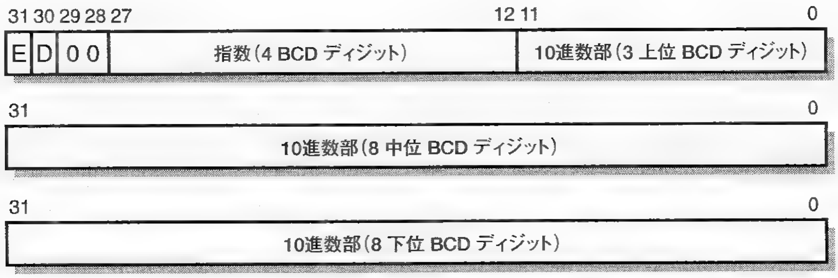
拡張倍精度形式を使えば，さらに精度が上がります．これは，3ワードに広がる80ビットの情報を使います．指数バイアスは16383，Jビットは小数点の左のビットです（正規化数の場合は'1'）．



〔図6.4〕IEEE754拡張倍精度の浮動小数点数形式

◆パック10進法

上で述べた2進法浮動小数点表示に加え，IEEE754規格はパック10進法形式を指定しています．式11を見ると，このパック形式でbは10，a，nは本章「10進法のバイナリ・エンコーディング」にあるようにバイナリ・エンコーディングした10進法でストアされています．数は正規化されているので$1 \leq a < 10$です．パック10進法表記は次のとおりです．



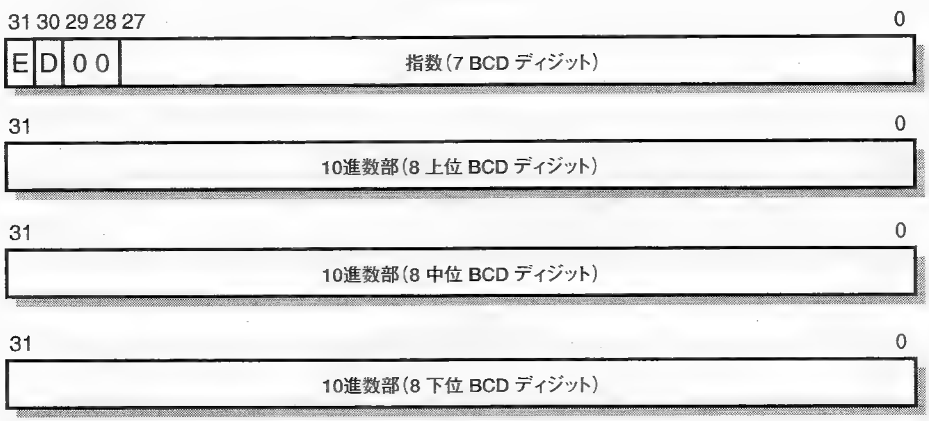
〔図6.5〕IEEE754パック10進法浮動小数点数形式

指数部の符号は最初のワードのビット31(「E」)にあり，10進法の符号はビット30(「D」)にあります．数の値は次のとおりです．

$$値(\text{packed}) = (-1)^{D} \times 10進数 \times 10^{((-1)^{E} \times 指数)} \qquad (式17)$$

◆拡張パック10進法

パック10進法形式は，より精度を上げるために4ワードを使っています．値は上の(式17)で求められ，形式は次のとおりです．



〔図6.6〕IEEE754拡張パック10進法浮動小数点数形式

◆ARM浮動小数点命令

標準のARM整数コアでは，上記の浮動小数点データ型はどれも直接サポートしていませんが，ARM Limitedはコプロセッサ命令空間内に浮動小数点命令セットを定義しています．これらの命令は通常，未定義命令トラップ(ハードウェア・コプロセッサによって受け入れられないすべてのコプロセッサ命令を集める)を通して完全にソフトウェアで実装されますが，サブセットはFPA10浮動小数点コプロセッサによってハードウェアで処理できます．

◆ARM浮動小数点ライブラリ

ARM浮動小数点命令セットの代替オプションとして（Thumbコードでは，これしか使えません），ARM LimitedはIEEE単および倍精度形式をサポートするCの浮動小数点ライブラリも供給しています．Cコンパイラにはソフトウェア・エミュレーションより速く（浮動小数点命令を受け取り，デコード，エミュレートする必要がなくなるため），コンパクトな（使用している関数のみイメージに入れればよいため）コードを生成するこの方法を選択するフラグがあります．

6.4　ARM浮動小数点アーキテクチャ

完全な浮動小数点サポートが必要な場合，ARM浮動小数点アーキテクチャは，前節で説明したデータ型についてより充実したサポートをします．完全にソフトウェアにおけるものもありますし，FPA10浮動小数点アクセラレータに関わるソフトウェア・ハードウェア混在ソリューションを使う場合もあります．

ARM浮動小数点アーキテクチャの特徴は次のとおりです．

(1) コプロセッサ番号が1または2のときのコプロセッサ命令セットの解釈（浮動小数点システムは二つの論理コプロセッサ番号を使う）

(2) コプロセッサ1および2における8本の80ビット浮動小数点レジスタ（同じ物理レジスタが両方の論理コプロセッサに現れる）

(3) さまざまな処理オプションを制御し，エラー状態を表示するユーザ可視浮動小数点ステータス・レジスタ（FPSR）

(4) オプションとして，浮動小数点制御レジスタ（FPCR）は，ユーザ不可視にできる．ハードウェア・アクセラレータに固有のサポート・ソフトウェアによってのみ使用される．

ARMコプロセッサ・アーキテクチャは，浮動小数点エミュレータ（FPE）ソフトウェア，FPA10と浮動小数点アクセラレータ・サポート・コード（FPASC）の組み合わせ，もしくは同じ命令セットをサポートする任意のハードウェア・ソフトウェアの組み合わせ，を交互に用いることができます．アプリケーション・バイナリは，どちらをサポートする環境でも動きますが，コンパイラ最適化の方針は異なります（FPEソフトウェアはグループ化された命令で，FPA10/FPASCは分散化された命令で最適に動作する）．

◆FPA10のデータ型

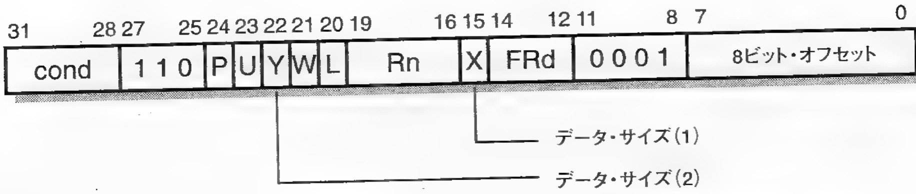
ARMのFPA10ハードウェア浮動小数点アクセラレータは単精度，倍精度，拡張倍精度形式をサポートしています．パック10進法形式はソフトウェアでのみサポートされます．

コプロセッサ・レジスタはすべて拡張倍精度形式で，内部的計算はすべて，このもっとも精度の高い形式で実行されます．例外は，フル80ビット精度を生成しない命令の高速バージョンがある場合です．しかし，メモリとレジスタ間のロードおよびストアで必要に応じて精度を変換することができます．

このことは，ARM整数アーキテクチャ内での整数の扱いに似ています．内部処理は32ビット長で実行しますが，メモリ転送ではバイトや時にはハーフ・ワードを指定することもできます．

◆浮動小数点命令のロードおよびストア

浮動小数点レジスタは8本しかないので，コプロセッサ・データ転送命令のレジスタ指定フィールド（図5.15のとおり）はスペア・ビットをもっています．スペア・ビットはここでは追加データ・サイズ指定に使用します．

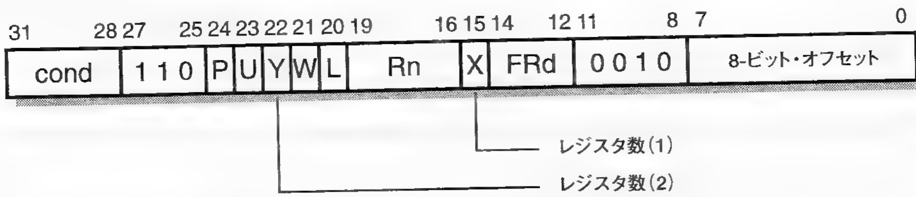


〔図6.7〕浮動小数点のロードおよびストアのバイナリ・エンコーディング

この形式の他フィールドは，5.18節に説明があります．XおよびYビットは4精度のうち一つ，つまり単精度，倍精度，拡張倍精度，パック10進法から選択して指定できます（パック10進法と拡張パック10進法のどちらを選ぶかはFPSRのビットが制御する）．

◆複数の浮動小数点レジスタのロードおよびストア命令

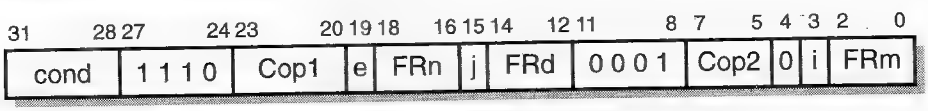
複数の浮動小数点レジスタのロードおよびストア命令は浮動小数点レジスタの状態の保存，復元に使います．各レジスタは3メモリ・ワードを使って保存され，正確な形式は定義されていません．つまり保存した値を使うのは，同等の複数の浮動小数点レジスタ・ロード命令を使って保存値を再ロードし，コンテクストを復元するときのみです．FRdは転送する最初のレジスタを指定，'X'，'Y' は1から4本の転送レジスタ数をエンコードします．ほかの浮動小数点命令がコプロセッサ番号1を使うのに対し，この命令はコプロセッサ番号2を使うことに注意してください．



〔図6.8〕複数の浮動小数点レジスタのロードおよびストアのバイナリ・エンコーディング

◆浮動小数点データの処理

浮動小数点データの処理は，浮動小数点レジスタ内の値に対して演算処理を実行します．外の世界との唯一の接点は，ARMコプロセッサ・ハンドシェイクを通して処理が完了したかを確認することです（たしかに，浮動小数点コプロセッサがハンドシェイク前に命令実行を始めることもあるが，そのときは状態変化をコミットする前にARMの承認を待つ）．



〔図6.9〕浮動小数点データ処理のバイナリ・エンコーディング

命令形式に多くのOPコード・ビットがあるのは，三つの各レジスタ指定フィールドからの追加ビットによって増補されているからです．これは，次のように8本の浮動小数点レジスタのうち一つを指定するには3ビットしか必要ないためです．

- iビット：レジスタ（FRm）または第2オペランド用の8定数のうち一つから選択
- eビット，Cop2ビット：デスティネーション・サイズと丸めモードを制御

- jビット：単項演算か2項演算の選択

命令には単純演算(加算，減算，乗算，除算，剰余，累乗)，超越関数(対数，指数，正弦，余弦，正接，逆正弦，逆余弦，逆正接)，その他(平方根，移動，絶対値，丸め)があります．

◆浮動小数点レジスタの転送

レジスタ転送命令は，ARMレジスタから値を受け取ったり値を戻したりします．これは通常，浮動小数点処理関数と組み合わされています．

31　28	27　24	23　21	20	19	18　16	15　12	11　8	7　5	4	3	2　0
cond	1 1 1 0	Cop1	L	e	FRn	Rd	0 0 0 1	Cop2	1	i	FRm

〔図6.10〕浮動小数点レジスタ転送のバイナリ・エンコーディング

ARMから浮動小数点ユニットへの転送は，「float」(ARMレジスタの整数を浮動小数点レジスタの実数に変換すること)と浮動小数点ステータス・レジスタおよび制御レジスタへの書き込みを含みます．逆方向では，「fix」(浮動小数点レジスタの実数をARMレジスタの整数に変換すること)とステータスおよび制御レジスタの読み取りを含みます．

浮動小数点比較命令は，この命令タイプのうちRdがr15の特殊ケースです．二つの浮動小数点レジスタを比較し，比較結果はARM CPSRのN，Z，C，Vフラグに戻します．そしてフラグが命令の条件的実行を直接制御します．

- Nは「より小さい」
- Zは「等しい」
- C，Vは，オペランドが「NaN」(非数)のときに発生する「順序なし(unordered)」比較結果など複雑な条件を示します．

◆浮動小数点命令の頻度

FPA10は，さまざまな浮動小数点命令の一般的な使用頻度に基づいて設計されています．頻度は，浮動小数点エミュレータ・ソフトウェアを使ってコンパイル済みプログラムを動作させて測定します．表6.1にその使用頻度をまとめています．

統計は，多くのロードとストア命令の実行が支配的であることを示しています．したがってFPA10は，これらを内部的演算処理と同時に処理することができるように設計されています．

〔表6.1〕
浮動小数点命令の使用頻度

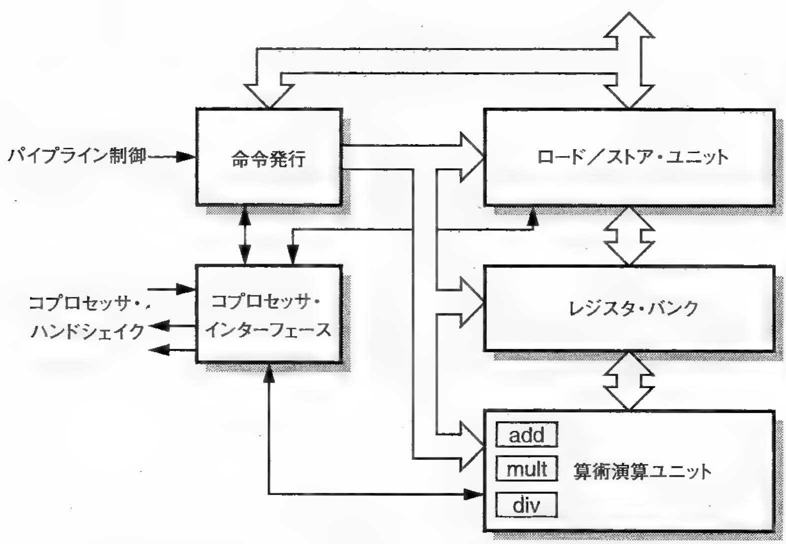
命令の種類	頻度
ロード／ストア	67%
加算	13%
乗算	10.5%
比較	3%
固定／浮動小数点	2%
除算	1.5%
その他	3%

◆FPA10の構成

FPA10の内部構成を図6.11に示します．外部インターフェースはARMデータ・バスとコプロセッサ・ハンドシェイク信号ですから，ピン数は比較的少なくてすみます．おもなコンポーネントは次のとおりです．

- コプロセッサ・パイプライン・フォロウ(4.5節)
- 浮動小数点データ型がメモリからロードされたりメモリにストアしたりする際にフォーマット変換を行うロード／ストア・ユニット
- 8本の80ビット拡張倍精度浮動小数点オペランドをストアするレジスタ・バンク
- 丸めと正規化のハードウェアとともに，加算器，乗算器，除算器を組み込んだ演算ユニット

ロード/ストア・ユニットは演算ユニットと同時に動作するため，前にロードしたオペランドが処理されている間に新しいオペランドをメモリからロードすることができます．ハードウェア・インタロックがデータ障害から保護します．



〔図6.11〕FPA10内部構成

◆FPA10パイプライン

FPA10アーキテクチャ・ユニットは4パイプライン・ステージで処理します．

(1) **準備**：オペランドの整列
(2) **計算**：加算，乗算，除算
(3) **アライン(整列)**：結果の正規化
(4) **丸め**：結果に適切な丸めを適用

浮動小数点処理は，命令パイプラインの中で検知されると同時に開始可能です(つまり，ARMハンドシェイクが起こる前に)が，結果のライトバックはハンドシェイクを待たなければなりません．

◆浮動小数点コンテクスト・スイッチ

FPAレジスタは，コンテクスト・スイッチにまたがって，保存と復元がされなければならない追加のプロセス状態を表します．しかし通常は，ほんの少数のアクティブなプロセスだけが浮動小数点命令を使います．

したがって，FPAレジスタの保存と復元をスイッチごとに行うのは不必要なオーバヘッドです．代わりに，ソフトウェアが次のアルゴリズムを使って保存と復元の数を最小化します．

(1) FPAを使う処理をスイッチ・アウトすると，FPAレジスタは保存しないがFPAをオフにする．
(2) 後述のプロセスか浮動小数点命令を実行するとトラップする．トラップ・コードはFPA状態を保存し，FPAをイネーブルする．

こうしてFPA状態の保存と復元のオーバヘッドは，FPAを使う処理にのみかかります．FPAを使う処理が一つしかない一般的なシステムでは，オーバーヘッドはすべて避けることができます．

◆FPA10のアプリケーション

FPA10は，ARM7500FEチップ上のマクロセルとして使用します(13.5節参照)．

◆VFP10

ARM10TDMIプロセッサ・コアで動作するために，もっと高性能な浮動小数点ユニットであるVFP10が設計されました(12.6節参照)．VFP10は，ベクタ浮動小数点演算を含めFPA10とは異なる浮動小数点命令セットをサポートします．

6.5 式

nビット整数の符号なし演算は，ANSI Cで「モジュロ2^n」と定義されていますから，桁あふれは起こりえません．したがって，基本的なARM整数データ処理命令はCの整数演算，ビット操作，シフト操作の多くを直接実現しています．例外は，除算と剰余で，これは複数のARM命令を必要とします．

◆レジスタの使用

データ処理命令はすべてレジスタの値に対してのみ実行されますから，複雑な式を効率的に評価するには，要求された値をレジスタに正しい順序で入れ，頻繁に使われる値が通常はレジスタにあるようにしなければなりません．これは，レジスタに保持できる値の数と，式評価中の中間値に使うレジスタの残り数とのトレードオフです(レジスタはメモリからフェッチするオペランドのアドレスを保持するのにも必要)．

このトレードオフを最適化するのは，おもにコンパイラの仕事です．コンパイラはオペランドをロードおよび組み合わせる正しい順序を割り出し，言語で定義された演算子優先順位が定めた結果を導きます．

◆ARMサポート

ARMが使用している3アドレス命令形式では，式評価の間どのようにレジスタをとっておいたり再利用したりするのかについて，コンパイラに最大限の裁量を与えます．

Thumb(サム)命令は通常2アドレスで，コンパイラの自由度が一定の程度にまで低下するうえ，汎用レジスタの数が少ないために，さらに処理が困難になっています(そして効率性の低いコードになる)．

◆オペランドのアクセス

手続きは通常，次の方法で示されるオペランドとともに動作し，アクセスされます．

(1) レジスタ経由で渡される引き数

値はすでにレジスタにあるため，これ以上の処理は不必要です．

(2) スタック上で渡された引き数

コンパイル時にわかる即値オフセット付きスタック・ポインタ(r13)相対アドレシングにより，単一LDRでオペランドが収集できます．

(3) 手続きのリテラル・プールの中の定数

コンパイル時にわかる即値オフセット付きPC相対アドレシングにより，単一LDRでアクセスできます．

(4) ローカル変数

ローカル変数にはスタック上の空間が割り当てられ，スタック・ポインタ相対LDRによりアクセスされます．

(5) グローバル変数

グローバル(と静的)変数にはスタティック領域の空間が割り当てられ，スタティック・ベース相対アドレシングによりアクセスされます．スタティック・ベースは通常r9にあります(本章後出の「ARM手続き呼び出し規格」参照)．

渡された値がポインタの場合，それが指す構造体内のオペランドにアクセスするために追加LDR(イミディエート・オフセット付き)が必要になるかもしれません．

◆ポインタ演算

ポインタに対する演算は，ポインタが指しているデータ型のサイズによって決まります．ポインタがインクリメントされると，データ項目のバイト・サイズを単位として変化します．

したがって

```
int  *p
p = p + 1 ;
```

これは，pの値を4バイト増加します．データ型のサイズはコンパイル時でわかっているので，コンパイラは適切な量で定数をスケーリングすることができます．変数がオフセットとして使用されていれば，実行時にスケーリングされなければなりません．

```
int  i = 4 ;
p = p + i ;
```

もしpがr0，iがr1に保持されていれば，pの変化は次のようにコンパイルします．

```
ADD   r0, r0, r1,  LSL  #2  ; scale r1 to int
```

データ型が2バイトの累乗でないサイズをもつ構造体のとき，小さな定数の乗算が必要になります．シフトおよび加算命令が求める積を少ない操作数で生成します．その際，必要に応じて一時レジスタを使います．

◆配列

Cの配列はポインタ処理の短縮表記のようなものですから，上のコメントはここでも当てはまります．宣言は次のとおりです．

```
int  a[10] ;
```

これにより*a*という名前が定義され，それは配列の最初の要素を指すただのポインタです．そしてa[i]への

参照は，ポインタ＋オフセット形式*(a+i)と等しくなります．この二つは互換性があります．

6.6 条件文

条件文は，テストのブール結果が真(あるいは偽)のとき実行します．Cでは，if...else(もし…さもなければ…)文およびスイッチ(Cのcase文)を含みます．

◆if...else

ARMアーキテクチャは，条件的に実行される文が小さいとき，条件的表現をとりわけ効率的にサポートします．たとえば，これは2整数の最大を求めるCの文です．

```
        if (a>b) c=a; else c=b;
```

変数a，b，cがレジスタr0，r1，r2にあるとき，コンパイルしたコードは次のように単純になります．

```
        CMP     r0, r1          ; if (a>b)...
        MOVGT   r2, r0          ;  ..c=a..
        MOVLE   r2, r1          ; ...else c=b
```

(この場合，Cコンパイラはもっと明確なシーケンスを生成することができる．

```
        MOV     r2, r0          ; c=a
        CMP     r0, r1          ; if (a>b)...
        MOVLE   r2, r1          ;  ...c=b
```

しかし，より複雑なif文には拡張できないため，一般的な議論においては無視する．)

if，elseシーケンスは，各命令に同じ条件をもった少数の命令でできています(ただし，条件的に実行されるどの命令も条件コードを変えないとする)が，2，3命令以上になる場合は従来のソリューションのほうが通常は便利です．

```
        CMP     r0, r1          ; if (a>b)...
        BLE     ELSE            ; skip clause if false
        MOV     r2, r0          ;  ..c=a..
        B       ENDIF           ; skip else clause
ELSE    MOV     r2, r1          ; ...else c=b
ENDIF   ..
```

ここではif，elseシーケンスは，どんな長さでもよく，自由に条件コードを使うことができます(たとえば，ネストされたif文)．なぜなら，それらのシーケンスは比較命令のすぐ後の分岐を飛び越す必要がないからです．

しかし，この単純な例で，どちらの分岐をとるにしても2番目のコード・シーケンスは最初のシーケンスより実行に約2倍の時間がかかることに注目してください．

分岐はARMにおいて高価ですから，最初のシーケンスに分岐がないことは非常に効率的です．

◆スイッチ

スイッチすなわちケース文は「if...else...」文の二者択一を多くの選択肢に拡張します．スイッチ文の標準的なC形式は次のとおりです．

```
switch (expression) {
  case constant-expression1: statements1
  case constant-expression2: statements2
  ...
  case constant-expressionN: statementsN
  default: statementsD
}
```

通常，文の各グループは「ブレーク」（または「リターン」）で終わり，スイッチ文を終了します．さもなければ，C言語の文法により次の文グループを続けて実行してしまいます．「ブレーク」のあるスイッチ文は，つねに同等の「if...else...」文に変換することができます．

```
temp = expression;
if (temp==constant-expression1) {statements1}
else ...
else if (temp==constant-expressionN) {statementsN}
else {statementsD}
```

しかし，スイッチ文が多くのケースをもつ場合，これは遅いコードになります．代わりに使えるのが「ジャンプ・テーブル」です．単純な形式のジャンプ・テーブルには，スイッチ式の取り得る各値のターゲット・アドレスが入っています．

```
                  ; r0 contains value of expression
        ADR       r1, JUMPTABLE      ; get base of jump table
        CMP       r0, #TABLEMAX      ; check for overrun..
        LDRLS     pc, [r1,r0,LSL #2] ;  .. if OK get pc
                  ; statementsD      ;  .. otherwise default
        B         EXIT               ; break
L1      ..        ; statements1
        B         EXIT               ; break
        ..
LN      ..        ; statementsN
EXIT    ..
```

明らかに，ジャンプ・テーブルが32ビット整数の取り得るすべての値についてのアドレスを含むことは不可能です．また，ジャンプ・テーブル・ルックアップがテーブルの端をいきすぎてしまわないことが重要なことも同様にはっきりしていますから，何らかのチェックが必要です．

スイッチ文をコンパイルするもう一つの方法は，次のように（効果的に）終了する「Dhrystone」ベンチマーク・プログラムの中の手続きで示されています．

```
  switch (a) {
    case 0: *b = 0; break;
    case 1: if (c>100) *b = 0; else *b = 3; break;
    case 2: *b = 1; break
    case 3: break;
    case 4: *b = 2; break;
  } /* end of switch */
} /* end of procedure */
```

生成されるコードは，ARM命令セットの多くの特徴を示しています．スイッチ文は，式の値をワード・オフセットにスケーリングした後，（v2のもの．レジスタ命名の慣習は，6.8節に説明するARMプロシージャ・

コール規格に従う)PCと加算することで実現します．オーバランのケースでは，加算命令(ADDLS)を飛ばしてPCオフセットにより空いたスロット命令(ここではLDMDB)が実行されます．命令が一つでよいケース(case3など)と，どんな長さでも最後のケースについてはすべて順番に完了します．ほかは分岐が必要です．この例は，条件命令を使って実現した「if...then...else」も示しています．

```
        ; on entry a1 = 0, a2 = 3, v2 = switch expression
        CMP     v2,#4               ; check value for overrun..
        ADDLS   pc,pc,v2,LSL #2     ;  .. if OK, add to pc (+8)
        LDMDB   fp,{v1,v2,fp,sp,pc} ;  .. if not OK, return
        B       L0                  ; case 0
        B       L1                  ; case 1
        B       L2                  ; case 2
        LDMDB   fp,{v1,v2,fp,sp,pc} ; case 3 (return)
        MOV     a1,#2               ; case 4
        STR     a1,[v1]
        LDMDB   fp,{v1,v2,fp,sp,pc} ; return
L0      STR     a1,[v1]
        LDMDB   fp,{v1,v2,fp,sp,pc} ; return
L1      LDR     a3,c_ADDR           ; get address of c
        LDR     a3,[a3]             ; get c
        CMP     a3,#&64             ; c>100?..
        STRLE   a2,[v1]             ;  .. No:  *b = 3
        STRGT   a1,[v1]             ;  .. Yes: *b = 0
        LDMDB   fp,{v1,v2,fp,sp,pc} ; return
c_ADDR  DCD     <address of c>
L2      MOV     a1,#1
        STR     a1,[v1]             ; *b = 1
        LDMDB   fp,{v1,v2,fp,sp,pc} ; return
```

6.7 ループ

C言語は3形式のループ制御構造をサポートしています．

- for(e1;e2;e3){..}
- while(e1){..}
- do{..}while(e1)

ここで，e1，e2，e3は「真」，「偽」を評価する式，{..}は制御構造が決定する回数だけ実行するループのボディです．ループのボディは簡単であることが多く，制御構造のオーバヘッドを最小化するため，コンパイラに負担をかけています．各ループは最低一つの分岐命令を持たなければなりませんが，それ以上は無駄になります．

◆forループ

典型的なforループは，インデクスを管理するために制御式を使っています．

```
        for (i=0; i<10; i++) {a[i] = 0}
```

最初の制御式はループ開始前に一度実行，2番目はループに入るごとにテストし，ループを実行するかどうかを制御します．3番目はループを通る各パスの終わりで実行し，次のパスに備えます．このループは次のようにコンパイルします．

```
        MOV     r1, #0              ; value to store in a[i]
        ADR     r2, a[0]            ; r2 points to a[0]
        MOV     r0, #0              ; i=0
LOOP    CMP     r0, #10             ; i<10 ?
        BGE     EXIT                ; if i >= 10 finish
        STR     r1, [r2,r0,LSL #2]; a[i] = 0
        ADD     r0, r0, #1          ; i++
        B       LOOP
EXIT    ..
```

(この例は「強度削減」の最適化テクニック，つまり固定した処理をループ外へ移動するテクニックを示している．最初の2命令は，Cプログラムにおいて論理的にはループ内にあるが，ループを回るたびに同じ定数にレジスタを初期化するだけなので外へ移動されている.)

コードを改良するには，条件分岐をとばしてEXITへ行き，次の命令に逆条件を適用して，プログラムが終了条件に到達したら元に戻るのではなく，通り抜けるようにします．しかし，テストをループの最後に移動すれば，さらに改良も可能です(この例でそれが可能なのは，初期化とテストが定数に関するもので，コンパイラはループが少なくとも1回は実行されることが確実にわかっているためである)．

◆whileループ

whileループは，構造が単純で，一般に反復の回数が一定でなく，ランタイムで変数によって明らかに定義される場合のループを制御します．「while」ループの標準的な概念的な配置は次のとおりです．

```
LOOP    ..                  ; evaluate expression
        BEQ     EXIT
        ..                  ; loop body
        B       LOOP
EXIT    ..
```

コードは，本体がまったく実行されない場合も許さなければならないので，分岐が二つ必要です．少し順序を変更すると，次のようにより効率的なコードになります．

```
        B       TEST
LOOP    ..                  ; loop body
TEST    ..                  ; evaluate expression
        BNE     LOOP
EXIT    ..
```

2番目のコード・シーケンスは，元のバージョンとまったく同様にループ・ボディを実行，制御式を評価し，コード・サイズも同じですが．反復ごとに(とらない)分岐が一つ減るため，2番目の分岐がループ・オーバヘッドから除かれます．次のように，より効率的なコード・シーケンスをコンパイラで生成することも可能です．

```
            ..                ; evaluate expression
            BEQ     EXIT      ; skip loop if necessary
    LOOP    ..                ; loop body
    TEST    ..                ; evaluate expression
            BNE     LOOP
    EXIT    ..
```

これが効率的なのは，完全なwhile構造に出会うたびに実行する分岐が1回少なくなることです（ボディは最低1回実行するとする）．これは利点が少ない割に余計な命令がかかるため，コード・サイズより性能を重視する場合にのみ使う価値があります．

◆「do...while」ループ

「do...while」ループの概念的な配置は，上の改良「while」ループと似ていますが，ループ・ボディをテスト前に実行するため最初の分岐がありません（したがって，必ず最低1回は実行する）．

```
    LOOP    ..                ; loop body
            ..                ; evaluate expression
            BNE     LOOP
    EXIT    ..
```

6.8　関数と手続き

◆プログラム設計

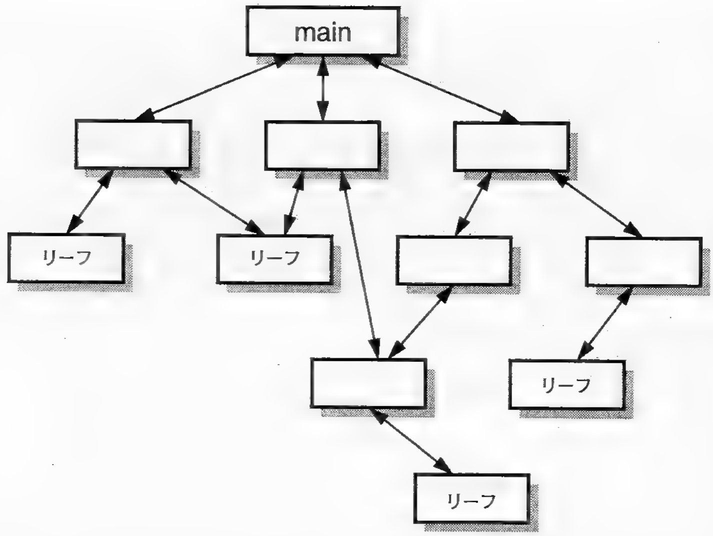
良いプログラム工程では，大きなプログラムは一つずつ完全にテストできる小さなコンポーネントに分割します．大きな一体構造のプログラムは複雑すぎて完全にテストできず，隠れた隅に「バグ」のある可能性があります．そして，バグの発見が遅すぎると，ユーザに発送される前のプログラムのライフサイクルの初期段階で直すことが不可能になります．

小さいソフトウェア部品のそれぞれは，きちんと定義されたインターフェースを使って，ある特定の動作を実行すべきです．それがどのように処理するのかは，プログラムの他の部分にとって重要ではありません（これが**抽象化**の原則である．6.1節を参照のこと）．

◆プログラム階層

また，完全なプログラムはソフトウェア部品の**階層構造**として設計され，たんなる平坦なリストではありません．

典型的な階層は，図6.12に示しました．階層のトップは「main」と呼ばれるプログラムです．残りの階層は，かなりフレキシブルに変化します．上層ルーチンが下層ルーチンを共用することもあれば，呼び出しが階層をスキップしたり，深さは階層内でさまざまです．



〔図6.12〕
典型的な階層プログラム構造

◆リーフ・ルーチン

階層の最下層には，**リーフ**(leaf：葉)ルーチンがあります．このルーチンは，それより下層のルーチンを呼びません．一般的なプログラムで最下層のルーチンにはライブラリや**システム**関数などがあります．これらは，前もって定義した処理であり，リーフであることもそうでないこともあります(つまり，内部構造をもっていることも，もっていないこともある)．

◆用語

プログラム構造を表すには，しばしばあいまいではありますが，いくつかの用語が使われます．本書では，それらの用語を次のように使用します．

- **サブルーチン**：上層ルーチンによって呼び出されるルーチンに対する一般的な用語．とくにプログラムをアセンブリ言語レベルで見るときに使用します．
- **関数**：名前を通して値を戻すサブルーチン．典型的な呼び出しは次のとおりです．

```
c = max (a, b) ;
```

- **手続き**：プロシージャともいいます．特定のデータ項目に関して処理を実行させるために呼び出すサブルーチン．典型的な呼び出しは次のとおりです．

```
printf  ("hello World\n") ;
```

◆Cの関数

関数と手続きを明確に区別するプログラミング言語もありますが，Cはそうではありません．C言語では，全サブルーチンが関数ですが，それらは値を戻すだけでなく副作用をもつことがあります．戻した値が`void`タイプのとき，値は実質的には返されず副作用だけが残り，まるで手続きのような動作をすることがあります．

◆引き数とパラメータ

引き数とは，関数呼び出しの際に渡される式のことをいいます．関数が受ける値はパラメータといいます．

C言語は，厳格な「値呼び出し」セマンティクスを用いていて，関数が呼ばれると各引き数のコピーが作られます．関数がパラメータの値を変えたとしても，これらは引き数のコピーなので，引き数自体は変わりません．(**参照呼び出し**では，関数内でパラメータに加えた変更は，呼んでいるプログラムへ戻される．これは引き数が単純な変数である場合のみ有効だが，C言語はこれをサポートしない．)

単一の値を戻すほかに，Cの関数で呼び出しているプログラム内のプログラムを変更できる方法は，データへのポインタを引き数として渡すことです．その場合，関数はポインタを使ってデータ構造にアクセス，変更します．

◆ARM手続き呼び出し(プロシージャ・コール)標準

アセンブリ言語で書かれたルーチンと，コンパイラで生成されたルーチンを柔軟に混在したものにも対応できるよう，ARM Limitedは手続きのエントリとイグジットのための規則を設定しています．ARMプロシージャ・コール標準(APCS)はARM Cコンパイラに採用されていますが，Cプログラマにとってこれが重要になるのはアセンブリ・レベルの出力を詳細まで理解しなければならないときだけです．

APCSは，次のように，特別な特徴のないARMアーキテクチャに多くの規則を課しています．

(1) 「汎用」レジスタに特定の用法を定義している．

(2) ARM命令セットがサポートするフル・エンプティ，上昇・下降などの選択肢の中からどの形式のスタックを用いるかを定義している．

(3) プログラムをデバッグする際，バックトレースに使用するスタック・ベースのデータ構造の形式を定義する．

(4) すべての外部的に可視な関数と手続きによって使用される関数引き数と結果の引き渡しメカニズムを定義している(「外部的に可視」とは，手続きインターフェースが現在のプログラミング・モジュールの外から提供されていることである．現在のモジュール内でのみ使われる関数は，この規則から外れて最適化される)．

(5) ARM共用ライブラリ・ルーチン・メカニズムをサポートしている．つまり，スタティックなデータをアクセスする共用(リエントラント)コードの標準的方法をサポートしていることを意味している．

◆APCSレジスタの用法

現在可視である16のARMレジスタの用法に関する規則は，**表6.2**にまとめてあります．レジスタは3セットに分類できます．

(1) 関数に値を渡す4本の引き数レジスタ

関数は，これらを保存する必要がありませんから，パラメータ値さえ使用または保存すれば，あとはスクラッチ・レジスタとして使用できます．関数が他関数に行う呼び出しをまたいでは保存されないので，再度必要な値を含んでいる場合は，コール間にまたがって保存しなければなりません．そのように使用されるときは「呼び出し元保存」レジスタ変数になります．

(2) 関数が値を変更しないまま戻さなければならない5本(7本まで)のレジスタ変数

これらは「呼び出し先保存」レジスタ変数です．関数がレジスタを使いたいときは保存しなければなりませんが，レジスタの変更は呼ばれる関数にかかっています．

(3) 少なくともある時点では専用の役割をもつ7本(5本まで)のレジスタ

たとえばリンク・レジスタ(r1)は，関数エントリに際してリターン・アドレスを運びますが，保存すれば(関数がサブ関数を呼ぶときは保存しなければならない)，あとはスクラッチ・レジスタとして使用できます．

〔表6.2〕APCSレジスタ使用規則

レジスタ	APCS名	APCSの役割
0	a1	引き数1／整数結果／スクラッチ・レジスタ
1	a2	引き数2／スクラッチ・レジスタ
2	a3	引き数3／スクラッチ・レジスタ
3	a4	引き数4／スクラッチ・レジスタ
4	v1	レジスタ変数1
5	v2	レジスタ変数2
6	v3	レジスタ変数3
7	v4	レジスタ変数4
8	v5	レジスタ変数5
9	sb/v6	スタティック・ベース／レジスタ変数6
10	sl/v7	スタック限界／レジスタ変数7
11	fp	フレーム・ポインタ
12	ip	スクラッチ・レジスタ／内部リンク・ユニット・コールの新sb
13	sp	カレント・スタック・フレームの最下部
14	lr	リンク・アドレス／スクラッチ・レジスタ
15	pc	プログラム・カウンタ

◆APCSのバリエーション

APCSには複数(16)のバリエーションがあり，さまざまなシステム向けにコードを生成します．サポートするのは次のものです．

(1) 32または26ビットPC

旧いARMプロセッサは26ビット・アドレス空間で動作していました．その後のバージョンの中にも後方互換性の理由から26ビットに対応するものがあります．

(2) 暗示的，明示的なスタック限界チェック

コードが高信頼性で動作するためには，スタック・オーバフローが検知されなければなりません．コンパイラはオーバフローの明示的チェックのために命令を挿入することができます．

メモリ管理ハードウェアが使える場合，ARMシステムではページ単位でメモリをスタックに割り当てることができます．次の論理ページがマップアウトされていると，スタック・オーバフローがデータ・アボートを起こすため検知できます．こうして，メモリ管理ユニットがスタック限界チェックを実行するため，明示的チェックのためにコンパイラが命令を挿入する必要はありません．

(3) 浮動小数点引き数を渡す二つの方法

ARM浮動小数点アーキテクチャ(6.4節)は8本の浮動小数点レジスタ・セットを指定します．APCSはこれらを使って浮動小数点引き数を関数に渡せますが，システムが浮動小数点変数を頻繁に使うときは，これがもっとも効率的な方法です．しかし，システムが浮動小数点タイプをほとんどまたはまったく使用しない場合(多くのARMシステムは使用しない)，この方法ではオーバヘッドが少しかかり，それを防ぐために浮動小数

点引き数を整数レジスタとスタック(もしくはその片方)で渡します.

(4) リエントラントまたは非リエントラント・コード

リエントラントと指定されたコードは位置独立性があり，スタティック・ベース・レジスタ(sb)経由で間接的にアドレスを指定します．このコードはROMに置くことがあり，いくつかのクライアント・プロセスによって共用されます．一般に，ROMまたは共用ライブラリにあるコードはリエントラントにすべきですが，アプリケーション・コードはそうではありません.

◆引き数渡し

Cの関数は多数の(ときには可変個の)引き数をもっています．APCSは，次のように引き数を構成します.

(1) 浮動小数点値が浮動小数点レジスタを通して渡されるとき，最初の四つの浮動小数点引き数を最初の四つの浮動小数点レジスタにロードする.

(2) 残りの引き数は，すべてワード・リストとして構成される．最初の4ワードはa1からa4へロード，残りのワードは逆順にスタックにプッシュする.

倍精度浮動小数点値など複数ワード引き数は，整数レジスタで渡されることもあります．スタックや，レジスタとスタックに分割することすらあります.

◆結果の戻し

単純な結果(整数など)は，a1で戻ります．複雑な結果は，メモリのアドレス指定された位置に入れて戻ります．アドレスは，追加第1引き数としてa1を通して関数へ渡されたものです.

◆関数のエントリとイグジット

a1からa4だけを使って実行する単純なリーフ関数は，わずかな呼び出しオーバヘッドのコードにコンパイルできます.

```
            BL      leaf1
            ..
    leaf1   ..
            MOV     pc, lr  ; return
```

一般的なプログラムでは，全関数呼び出しの50%がリーフ関数に対するもので，きわめて単純です.

レジスタが保存されなければならない場合，関数はスタック・フレームを作らなければなりません．これは，ARMの複数のロードおよびストア命令を用いて効率的にコンパイルできます.

```
            BL      leaf2
            ..
    leaf2   STMFD   sp!, {regs, lr}    ; save registers
            ..
            LDMFD   sp!, {regs, pc}    ; restore and return
```

ここで保存，復元するレジスタ数は関数を実装するのに必要な最小限数です．リンク・レジスタ(lr)から保存した値は直接プログラム・カウンタ(pc)へ戻るため，lrは関数のボディでスクラッチ・レジスタとして使用することができます(上記の単純なケースにはない).

スタックのバックトレース・データ構造の作成，浮動小数点レジスタに渡した浮動小数点引き数の処理，スタック・オーバフローのチェックなどが必要な場合は，より複雑な関数エントリ・シーケンスを使います.

◆末尾継続関数

復帰直前にほかの関数を呼ぶ単純な関数は，通常，呼び出しオーバヘッドがあまりかかりません．つまりコンパイラが，継続する関数から直接コードを復帰させるのです．これによりベニヤ関数(たんに引き数を並び換えたり，型を変更したり追加引き数を足したりする関数)が非常に効率的になります．

◆ARMの効率性

全体としてARMは，関数や手続きを効率よく柔軟にサポートしています．手続き呼び出し標準のさまざまな性質がアプリケーション条件によく合い，どんな場合にも効率的なコードを生成します．複数レジスタのロードおよびストア命令は，このコンテクストでよい効果を生み出しています．この命令なしでは，非リーフ関数の呼び出しはもっと非効率的になっていたでしょう．

コンパイラにより，リーフや末尾継続関数の効果的な最適化が可能になり，効率的なプログラミング・スタイルを促進します．プログラマは，リーフ関数が効率的な場合にはより構造的な設計を用い，ベニヤ関数が効率的な場合は抽象化を利用することができます．

6.9 メモリの使用

多くのコンピュータ・システムと同様，ARMシステムでもメモリは論理アドレスの線形セットとして構成されています．Cプログラムは，プログラム・メモリの固定領域(アプリケーション・イメージがある場所)と二つのデータ領域をサポートするメモリにアクセスします．これら2領域は動的に拡大し，コンパイラが最大サイズを決められない場合もあります．これらの動的データ領域は次のものです．

(1) スタック

(小さくない)関数が呼ばれると，つねに新しいスタック・フレームをスタック上に生成し，そこにバックトレース記録，ローカル(非静的)変数などを置きます．関数が復帰するとスタック空間は自動的に消滅し，次の関数呼び出しに再利用されます．

(2) ヒープ

ヒープとは，プログラム要求(`malloc()`)が新しいデータ構造のための追加メモリを要求するのを満たすためのメモリ領域です．長期間にわたってメモリを要求し続けるプログラムについては，必要でなくなった全セクションを解放するように注意してください．さもなければ，メモリがなくなるまでヒープが拡大し続けます．

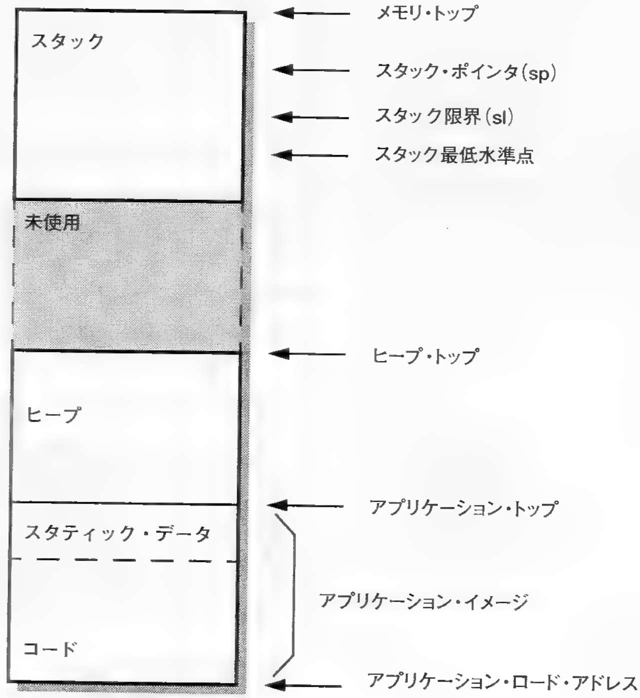
◆アドレス空間モデル

通常のメモリの使用については，図6.13に示します．アプリケーションがメモリ空間全体を使用できるとき(メモリ管理ユニットが，アプリケーションにメモリ空間全体が使用できると思わせているとき)，アプリケーション・イメージは，最下位アドレスにロードされ，ヒープはアプリケーション・トップから上へ，スタックはメモリ・トップから下へ伸びます．ヒープのトップとスタックのボトム間の未使用メモリは，必要に応じてヒープまたはスタックに割り当てられ，なくなるとプログラムがメモリ不足で停止します．

メモリ管理された一般的なARMシステムでは，一つのアプリケーションに割り当てた論理空間が非常に大きくなることがあり，その範囲は1～4Gバイトにもなります．メモリ管理ユニットは，要求があれば，割り当てるページがなくなる(物理メモリをすべて割り当てる，または仮想メモリをもつシステムではハード・ディスクのスワップ空間がなくなる)まで，ヒープやスタックにページを追加します．通常，ヒープのトップとス

タックのボトムが当たるまでに，かなり時間がかかります．

メモリ管理サポートをもたないシステムでは，OSの条件さえ満たせば，残りの物理メモリ・アドレス空間すべて(ほかのアプリケーションが同時に動作していない場合)，または一部(二つ以上のアプリケーションが動作している場合)がアプリケーションに割り当てられます．そして，ヒープのトップとスタックのボトムが出会ったちょうどそのときが，アプリケーション・メモリ不足になるときです．



〔図6.13〕
標準的なARM Cプログラム
アドレス空間モデル

◆チャンク・スタック・モデル

チャンクされているスタック，つまりスタックがヒープ内のチャンク(まとめられた固まり)の連続として実現される場合など，ほかのアドレス空間モデルもあります．これにより，アプリケーションが単一の隣接したメモリ領域を占めることが可能になり，そのメモリ領域は必要に応じて1方向に伸びるため，メモリが非常に少ないときなどは便利です．

◆スタックのふるまい

プログラム動作時のスタックのダイナミックなふるまいを理解することは重要です．なぜなら，それによりローカル変数(スタック上に割り当てられている空間)のスコープ規則が見えてくるからです．

下の単純なプログラム構造を考えてください．

```
main () {
  ..                      /* t1 */
  func1 ();
  ..                      /* t5 */
```

```
    func2 ();
    ..                      /* t7 */
  } /* end of main */

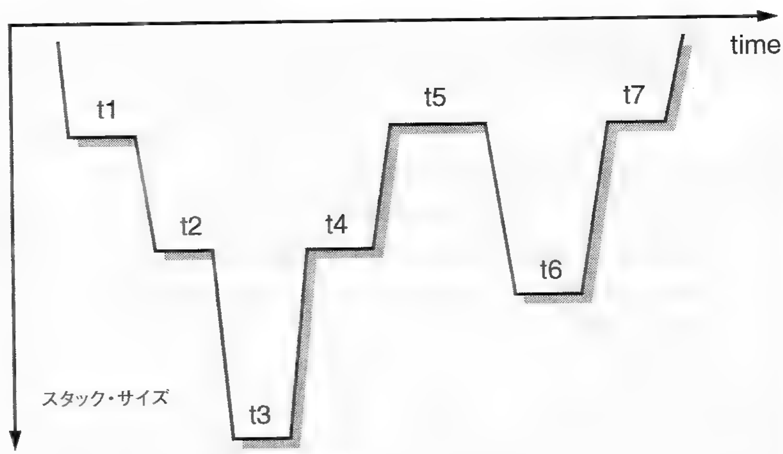
  func1 () {
    ..                      /* t2 */
    func2 ();
    ..                      /* t4 */
  } /* end of func1 */

  func2 () {
    ..                      /* t3, t6 */
  } /* end of func2 */
```

コンパイラが各関数呼び出しにスタック空間を割り当てると，スタック動作は図6.14に示すようになります．各関数呼び出しで，スタック空間が引き数に割り当てられ（引き数がすべてレジスタに渡されない場合），関数内で使うレジスタの保存，リターン・アドレスや旧いスタック・ポインタの保存，スタック上のローカル変数用メモリ割り当てなどに使用します．

関数イグジットでどのようにスタック空間が回復し，続く呼び出しで再使用されるか，また，どのようにfunc2( )への二つの呼び出しが異なるメモリ領域に割り当てられているか（図6.14におけるt3，t6のとき）に注意してください．したがって，もし最初のコール中のローカル変数に使われるメモリが他の関数の呼び出しによって上書きされなかったとしても，そのアドレスがわからないので，2番目のコール中ではアドレスはアクセスできません．いったん手続きから出ると，ローカル変数値は永久に失われます．

〔図6.14〕
スタック動作例



◆データ記憶

Cでは多くのデータ型がサポートされていますが，データ型によりバイナリ表現をストアするために必要なメモリ量が異なります．基本的なデータ型はバイト（文字），ハーフ・ワード（短整数），ワード（整数，単精度浮動小数点），複数ワード（倍精度浮動小数点）のうちいずれかを占めます．派生データ型（構造体，配列，ユニオンなど）は基本データ型の観点から定義されます．

多くのRISCプロセッサと同様，ARM命令セットはデータ項目が適切にメモリ内に整列されていれば，もっ

とも効率よくロードおよびストアを行います．1バイトのアクセスは，どんなバイト・アドレスでも同じ効率ですが，ワード・アラインされていないアドレスへ1ワードのストアするのは非常に非効率的で，七つのARM命令と一時的なワーク・レジスタを必要とします．

◆データ・アラインメント

したがって，ARM Cコンパイラは，通常，データ項目を適切な境界で整列（アライン）しています．

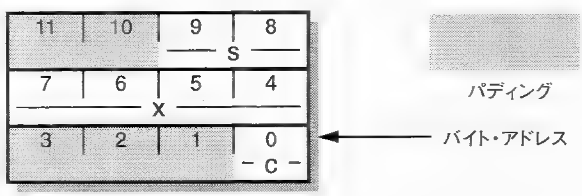
- バイトは任意のバイト・アドレスにストア
- ハーフ・ワードは偶数バイト・アドレスにストア
- ワードは4バイト境界上にストア

異なるタイプのデータ項目が同時に宣言された場合，コンパイラは整列させるために必要な箇所に「パディング（padding）」を用います．

```
struct S2 {char c; short s; int x;} example2;
```

この構造体は，図6.15のように3ワードのメモリを占めます〔構造体にはワード境界で終わるように最後にもパディング（詰めもの）が付く〕．

配列は，適切な基本データ項目を繰り返し，各アイテムのアラインメント規則に従うことによって，メモリ内でレイアウトします．

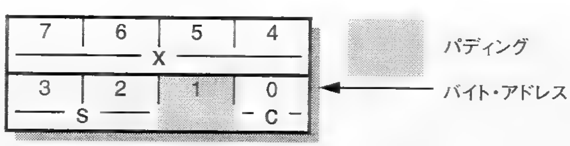


〔図6.15〕
通常の構造体のメモリ配置例

◆メモリ効率

上記に説明したデータ・アラインメント規則があれば，プログラマはコンパイラを使って構造体を適切に構成することによってメモリの無駄を最小限にすることができます．上と同じ内容をもつ構造体を下のように構成しなおすと，その場合元の3ワードではなく2メモリ・ワードしか使いません．

これにより図6.16のようなメモリ配置になります．一般的に，構造体要素を並び換えてワードより小さな型がワード内でグループ化されるようにすると，コンパイラが効率的なアラインメントを維持するために挿入するパディングの数が最少になります．



〔図6.16〕
より効率的な構造体のメモリ配置例

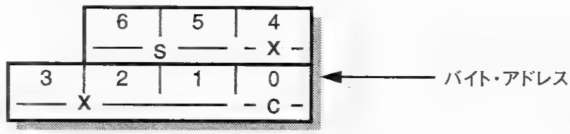
◆パックされた構造体

異なるアラインメント規則に従ったコンピュータとデータ交換したり，性能が低下してもメモリ使用を最小化するためにデータを詰め込んだりすることが必要な場合があります．そのような場合，ARM Cコンパイラはパディングをすべて取り去ったパックされたデータ構造で動作するコードを生成します．

```
__packed struct S3 {char c; int x; short s;} example3;
```

__packed structは全フィールドのアラインメントについて精密な制御をしますが，アラインしていないオペランドに対するアクセスはARMにおいて比較的非効率的なため，オーバヘッドがかかります．したがって，厳密に必要な場合のみ使わなければなりません．上で宣言したパックされた構造体のメモリ配置は図6.17にあります．

〔図6.17〕
パックされた構造体のメモリ配置例



6.10 ランタイム環境

Cプログラムは，動作するための環境を必要とします．これは通常，Cプログラムが呼ぶことのできる関数ライブラリによって提供されます．PCやワークステーションには，たいてい完全なANSI Cライブラリがあります．これにより，ファイル管理，入出力(printf())，リアルタイム・クロックなど幅広い関数にアクセス可能です．

◆最小ランタイム・ライブラリ

携帯電話など小さな組み込み型システムでは，これらの関数はほとんど重要ではありません．ARM Limitedは最小限のスタンドアローン・ランタイム・ライブラリを提供しており，これをターゲット環境にポートすれば基本的なCプログラムを動作することができます．したがって，このライブラリはCプログラムの最低条件を反映し，次のもので構成されています．

(1) 除算，剰余関数

ARM命令セットは除算命令を含まないため，ライブラリ関数として実現されています．

(2) スタック限界チェック関数

小さな組み込み型システムは，スタック・オーバフロー検知に使うメモリ管理ハードウェアをもっていないことがほとんどです．したがって，プログラムが安全に動作するためには，このライブラリ関数が必要です．

(3) スタックおよびヒープ管理

Cプログラムはすべて(多数の)関数呼び出しにスタックを使い，よほど小さなもの以外はヒープ上にデータ構造を作成します．

(4) プログラム・スタートアップ

スタックとヒープが初期化されると，main()への呼び出しからプログラムは開始します．

(5) プログラム終了

多くのプログラムは_exit()を呼んで終了します．永久に動作するプログラムでもエラーが検知されれば終了しなければなりません．

以上の最小ライブラリのために生成されたコードの合計サイズは736バイトです．そして，リンカが参照しないセクションをとばすので，ライブラリ・イメージはたいていの場合0.5Kバイト程度にまで低減するように実装されます．これにより，完全なANSI Cライブラリよりかなり小さくなります．

6.11　例題と演習

<例題6.1>　C言語で「Hello World」プログラムを書き，コンパイルし実行させなさい．

次のプログラムは必要な関数をもっています．

```
/* Hello World in C */
#include <stdio.h>

int main()
{
  printf( "Hello World\n" );
  return( 0 );
}
```

この例の重要事項は以下のとおりです．

- 「#include」指示により，このプログラムがCで使用可能なすべての標準的入出力関数を使用できるようになります．
- 「main」手続きの宣言．すべてのCプログラムは，これを呼び出すことによって動作するので必要です．
- 「printf(..)」ステートメントはstdioにある関数を呼び出し，出力を標準的な出力装置に送ります．デフォルトでは，ディスプレイ・ターミナルになっています．

アセンブリ・プログラミング演習で最大の難関は，テキスト編集から，コンパイル，リンク，プログラムの動作まで，ツールを通して作業フローを確立することです．このプログラムさえ動けば，複雑なプログラムを生成することも簡単です（少なくともプログラムの設計自体が困難になるほど複雑にならないかぎり）．

ARMソフトウェア開発ツールを使って，上のプログラムを「HelloW.c」として保存してください．そして，プロジェクト・マネージャとこのファイルを使って（プロジェクトの唯一のファイルとして）新しいプロジェクトを作成します．「Build」ボタンをクリックすると，プログラムはコンパイル，リンクされ，「Go」ボタンを押すとARMulator上で動作します．うまくいけば，ターミナル・ウィンドウに期待する出力があるはずです．

演習6.1.1　コンパイラから（「_s」オプションを使って）アセンブリリストを生成し，できたコードを見なさい．

演習6.1.2　デバッガでプログラムを動作し，シングル・ステップでプロセッサの進行をコードで観察しなさい．

<例題6.2>　32ビット・バイナリ，バイナリ・コード10進法，ASCII，単精度浮動小数点表記のそれぞれで数字2001を書きなさい．

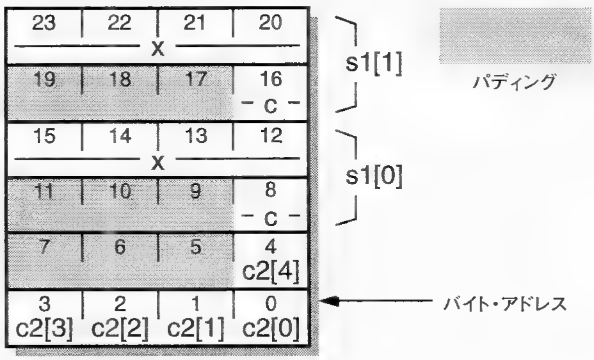
```
Binary: 2001    = 1024 + 512 + 256 + 128 + 64 + 16 + 1
                = 00000000000000000000011111010001 (base 2)
BCD:    2001    = 0010 0000 0000 0001
ASCII:  2001    = 00110010 00110000 00110000 00110001
F-P:    2001    = 1.1111010001 x 2^1010
                = 01001001 11110100 01000000 00000000
```

演習6.2.1　ローマ数字で書かれたデータを10進法形式に変換するCプログラムを書きなさい．

＜例題6.3＞ 次のデータがメモリ内でどのように構成されているかを示しなさい．

```
struct S1 {char c; int x;};
struct S2 {
  char c2[5];
  S1   s1[2];
} example;
```

最初のstruct文は型を定義するだけですから，メモリは割り当てられません．2番目は「example」と呼ばれる構造体を宣言します．これは，5文字の配列の後にS1型の二つの構造体の配列が続くものです．構造体は，ワード境界ではじまらなければならないので，文字配列は2ワードを占めるようにパディングします．したがって，メモリ構成は下のようになります．



演習6.3.1 パックされていたとすると，同じ構造体がどのようにメモリ内で構成されるかを示しなさい．

第7章 Thumb(サム)命令セット

Thumb(サム)命令セットは，コード密度の問題に対応するものです．ARM命令セットのサブセットの圧縮形式と考えてもかまいません．Thumb命令セットは，ARM命令セット上にマップし，Thumbプログラマ・モデルはARMプログラマ・モデル上にマップします．Thumb命令は，ARM命令パイプライン内の動的伸張を用いて実装され，標準プロセッサ内ではARM命令として実行します．

Thumbは完全なアーキテクチャとはいえません．つまり，ARM命令セットをサポートせずにプロセッサがThumb命令をサポートすることはありえないからです．したがってThumb命令セットは，一般的なアプリケーションの機能をサポートし，必要なときは完全なARM命令セットに頼ることができます(たとえば，例外はすべて自動的にARMモードに入る)．

Thumbは，ARM開発ツールによって完全にサポートされており，アプリケーションはARMとThumbのサブルーチンを柔軟に混ぜ，性能やコード密度をルーチンごとに最適化することが可能です．

本章では，Thumbアーキテクチャとその実装に触れ，Thumbの使用によりメリットが得られるアプリケーションの特徴を考えます．適切なアプリケーションでThumbを使えば，電力効率向上，コスト削減，性能拡大が一度に可能になります．

7.1 CPSRのThumb(サム)ビット

Thumb命令セットをサポートするARMプロセッサは，もちろん標準の32ビットARM命令セットを実行することができます．そして，命令ストリームの解釈は，つねにCPSRのビット5，つまりTビットが決定します(図2.2参照)．Tがセットされていれば，プロセッサは命令ストリームを16ビットThumb命令として解釈します．それ以外では，標準のARM命令として解釈されます．

すべてのARMプロセッサがThumb命令を実行できるわけではありません．9.1節で述べるように，ARM7TDMIなど名前にTの付くプロセッサだけです．

◆Thumbエントリ

ARMコアは，リセットの後に起動して，ARM命令を実行します．通常，ARMコアは分岐とexchange命令(BXのこと．5.5節を参照)を実行することによって，Thumb命令を実行するようスイッチします．命令セットは，指定されたレジスタの最下位ビットがセットされていればTビットをセットし，プログラム・カウンタをそのレジスタの残りのビットで与えられるアドレスにスイッチします．命令は分岐を起こすため，命令パイプラインをフラッシュし，すでにパイプラインにある命令の解釈に関するあいまいさを取り除きます(たんに実行しない)．

ARMからThumbコードに変更する命令には，例外復帰があります．データ処理命令の特別形式を用いる場合も，複数レジスタ・ロード命令の特別形式を用いる場合も同じです(第5章の「例外復帰」を参照)．どちらの命令も，例外に入る前に実行していた命令ストリームに復帰するのに一般的に用いられており，Thumbモードに計画的にスイッチするようには意図されていません．BXと同様，これらはプログラム・カウンタを変更，したがって命令パイプラインをフラッシュします．

◆Thumbイグジット

ARM命令ストリームへの明示的なスイッチバックは，7.3節にあるようにThumbのBX命令を実行することによって起こります．

ARM命令ストリームへの暗黙の復帰は，例外のたびに起こります．なぜなら例外エントリはつねにARMコードによって処理されるからです．

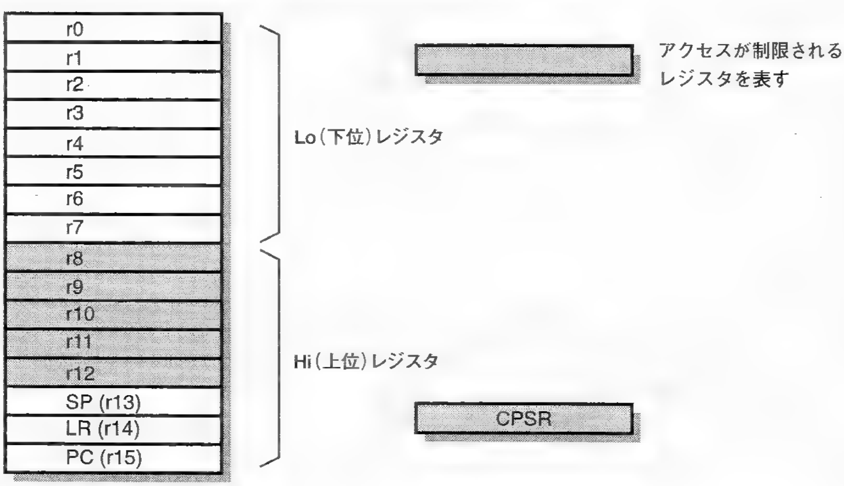
◆Thumbシステム

上記から，すべてのThumbシステムが，たとえ初期化と例外エントリ処理だけにしても，いくらかのARMコードを含むことが明らかです．

しかし，多くのThumbアプリケーションは，それ以上にARMコードを利用しています．典型的な組み込み型システムは，ARMコアと同じチップ上に高速32ビット・メモリを少量もっており，速度クリティカルなルーチン(ディジタル信号処理アルゴリズムなど)をこのメモリからARMコードで実行します．大部分のコードは普通，速度クリティカルではないので，16ビット・オフチップROMから実行できます．このことは本章の終わりで詳しく述べます．

7.2 Thumb(サム)プログラマ・モデル

Thumb命令セットはARM命令セットのサブセットであり，命令は限定されたARMレジスタで処理されます．プログラマ・モデルを図7.1に示します．命令セットは8本の「Lo」汎用レジスタr0からr7への完全なアクセスが可能で，以下のようにr13からr15を特殊な目的のために拡張に使用します．



〔図7.1〕
Thumbアクセス可能レジスタ

- r13をスタック・ポインタとして使用
- r14をリンク・レジスタとして使用
- r15をプログラム・カウンタ(PC)として使用

これらの用法は，ARM命令セットがレジスタを使う方法に非常に似ていますが，r13をスタック・ポインタとして使うのはARMコードでは純粋にソフトウェア規則であるのに比べ，Thumbコードでは多少ハードワイヤードになります．残りのレジスタ(r8からr12とCPSR)は，アクセスが次のように制限されています．

- わずかな命令のみが「Hi」レジスタ(r8からr15)を指定できる．
- CPSR条件コード・フラグは算術および論理演算によりセットされ，条件的分岐を制御する．

◆ThumbとARMの類似点

全Thumb命令は16ビット長です．ARM命令上にマップするため，ARM命令セットの性質を多く受け継いでいます．

(1) データ処理，データ転送，フロー制御命令をもつロード/ストア・アーキテクチャ
(2) ハーフ・ワードが2バイト境界で，ワードが4バイト境界でアラインされているとき，8ビット・バイト，16ビットハーフ・ワード，32ビット・ワード・データ・タイプのサポート
(3) 32ビットのセグメント化していないメモリ

◆ThumbとARMの相違点

しかし，16ビット命令長に圧縮するために，ARM命令セットの以下の特徴がなくなりました．

(1) ほとんどのThumb命令は無条件に実行される(ARM命令はすべて条件実行可能)．
(2) 多くのThumbデータ処理命令は2アドレス形式を使う(デスティネーション・レジスタはソース・レジスタの一つと同じ)．
(ARMデータ処理命令は，64ビット乗算を除いては3アドレス形式を用いる．)
(3) Thumb命令形式は，エンコーディング密度が高いためARM命令形式ほど規則的でない．

◆Thumb例外処理

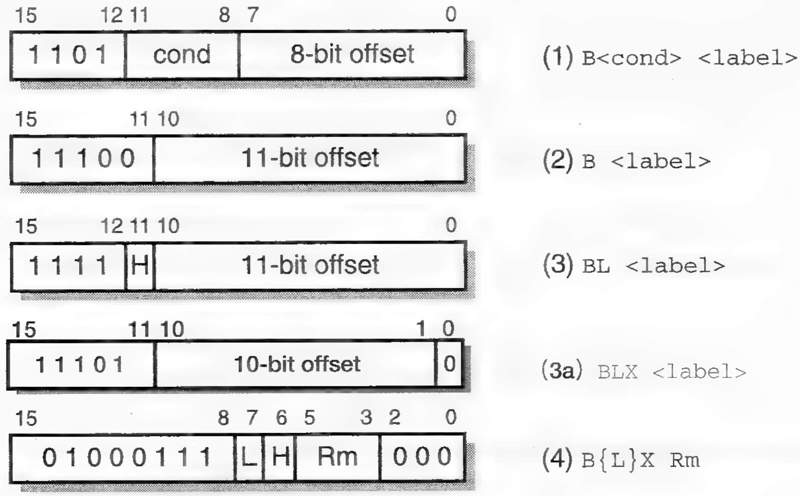
例外処理はすべて，プロセッサをARM実行に戻し，ARMプログラマ・モデルで処理します．TビットはCPSRにあるため，例外エントリの際は適切なSPSRに保存されます．そして，例外命令からの同じ復帰命令でプロセッサ状態を復元し，例外が最初に起こったときの状態のとおりARMまたはThumb命令を実行する状態にします．ARM例外復帰命令は，第5章の「例外復帰」で説明したように，ARMパイプライン動作を埋め合わせるために復帰アドレスの調整が必要です．Thumb命令は4バイト長ではなく2バイト長なので，Thumb実行から例外に入ったときの自然なオフセットは4バイト長の場合とは違ってきます．例外モードのリンク・レジスタにコピーしたPC値は，4バイトでなく2の倍数でインクリメントされているからです．しかし，Thumbアーキテクチャでは，復帰シーケンスが複雑にならないように，同じ復帰命令がどちらでも動作するように．リンク・レジスタ値がARM復帰オフセットと一致するように自動的に調整される必要があります．

7.3　Thumb(サム)分岐命令

フロー制御命令に含まれるのは，ARM命令セットに見られるPC相対分岐とリンク付き分岐命令の様々な形

式，そしてARM命令セットとThumb命令セットの間をスイッチするexchange交換命令です．

ARM命令には，明らかに16ビット命令形式には合わない大きな（24ビット）オフセット・フィールドがあります．そのため，Thumb命令セットは，機能性をサブセット化する方法を多くもっています．



〔図7.2〕
Thumb分岐命令の
バイナリ・エンコーディング

<<概要>>

分岐命令の典型的な用法は，次のとおりです．

(1)　短い条件的分岐で（たとえば）ループの出口を制御する

(2)　中距離の無条件分岐で，コードのgoto部分に対応する

(3)　長距離のサブルーチン呼び出し

ARMは上記すべてを同じ命令で処理しますが，とくに最初の二つの例では24ビット・オフセットの多くのビットが余分となります．Thumbは，図7.2に番号を付けたように，各部分に異なる形式を用いて効率よく処理する必要があります．

最初の二つの書式は，条件フィールドをオフセット長にトレードオフすることを示しています．最初の形式の条件フィールドはARM命令のものと同じです（5.3節を参照）．どちらの場合でもオフセットは左へ1ビット（ハーフ・ワード・アラインメントにするため）シフトし，32ビットへ符号拡張しています．

3番目の形式は，よりささいなものです．リンク付き分岐サブルーチン・メカニズムは，16ビット命令形式では困難な長い範囲をしばしば必要とします．そのため，Thumbは16ビット命令形式で2命令を使用し，ともにこのフォーマットで組み合わせて22ビットのハーフ・ワード・オフセット（32ビットに符号拡張される）にします．したがって命令の範囲は+/−4Mバイトです．二つの命令を独立したものであるようにするため，たとえばその命令の間でも割り込み処理できます．リンク・レジスタが一時的記憶として使われます．これは，結局二つの命令の後で上書きされますから，有用なものは何も格納できません．命令ペアの動作は次のとおりです．

```
1. (H=0)    LR := PC + (sign-extended offset shifted left 12 places);
2. (H=1)    PC := LR + (offset shifted left 1 place);
            LR := oldPC + 3.
```

ここで，「oldPC」は2番目の命令のアドレスです．つまり，リターン・アドレスは2バイト加えて次の命令をポイントするようにしてあり，また最下位ビットはThumbルーチンが呼んでいることを示すようセットして

あります．

書式3aも，BLXバリアントを作る場合の第2ステップです．これはアーキテクチャversion5Tにしかありません．第1ステップは上記のBLと同じものを使用します．

```
1. (BL, H=0) LR := PC + (sign-extended offset shifted left 12 places);
2. (BLX)     PC := LR + (offset shifted left 1 place) & 0xfffffffc;
             LR := oldPC + 3;
             the Thumb bit is cleared.
```

ターゲットがARM命令ですから，オフセットに必要なのは10ビットだけです．計算されたPC値ではビット[1]がまだセットされている可能性がありますが，暗黙的にクリアされることに注意してください．

4番目の形式は，直接ARM BX命令に対応しています（5.5節を参照）．ただし，BLXでは（version5Tでのみ使用可）r14が次の命令のアドレスプラス1にセットされていて，Thumbルーチンが呼んでいることを示しています．ここで「H」は「Hi」レジスタ（r8からr15）を選択するようセットできます．

<<書式>>

```
B<cond> <label> ; 書式1  - Thumb target
B       <label> ; 書式2  - Thumb target
BL      <label> ; 書式3  - Thumb target
BLX     <label> ; 書式3a - ARM target
B{L}X   <label> ; 書式4  - ARM or Thumb target
```

リンク付き分岐は，書式3の命令を生成します．書式3の命令を個別に使用することは考えてありません．これらは，つねにペアです．同様にBLXは書式3の命令と書式3aの命令を生成します．

アセンブラは，現在の命令アドレス，ターゲット・ラベルのアドレス，パイプライン動作の小さな修正などから，命令に挿入するべき妥当なオフセットを計算します．ターゲットが範囲から出ていれば，エラー・メッセージが出力されます．

<<同等のARM命令>>

書式1から3は，ARMの分岐およびリンク付き分岐命令に非常に似ていますが，ARM命令がワード（4バイト）オフセットのみをサポートするのに対し，Thumb命令はハーフ・ワード（2バイト）オフセットを要求します．したがって，これらのThumb命令からARM命令セットへの直接対応するものはありません．Thumbをサポートする ARMコアは，ハーフ・ワード分岐オフセットをサポートするよう多少修正されており，ARM分岐命令は偶数ハーフ・ワード・オフセットにマッピングされています．

書式4は，同じアセンブラの構文規則をもつARM命令と同等です．BLXは，アーキテクチャversion5Tを実装したARMプロセッサでのみサポートされています．

◆サブルーチン・コールとリターン

上記の命令とそれに相当するARM命令により，呼び出し側と同じもしくは反対の命令セットで書かれた関数へのサブルーチン・コールが可能となります．

同じ命令セットから呼ばれる関数だけが，従来のBLコール，それにMOV pc, r14またはLDMFD sp!{.., pc}(Thumbコードでは POP{.., pc})リターン・シーケンスを使用できます．

逆の命令セットから呼ばれた関数，またはどちらかの命令セットから呼ばれた関数は，BX lrまたはLDMFD sp!{.., rN}; BX rN(Thumbコードでは，POP {.., rN};BX rN)で復帰することができます．

アーキテクチャversion5TをサポートするARMプロセッサは，LDMFD sp!{...pc}(Thumbコードでは，POP {..pc})でも復帰することができます．これらの命令は，ロードしたPC値の最下位ビットを使ってThumbビットを更新しますが，これはversion5Tより前のアーキテクチャではサポートされていません．

7.4 Thumb(サム)ソフトウェア割り込み命令

Thumbソフトウェア割り込み命令は，ARMのものとまったく同様に動作し，例外エントリ・シーケンスがプロセッサをARM実行にスイッチします．



〔図7.3〕Thumbソフトウェア割り込みのバイナリ・エンコーディング

<<概要>>

この命令により次の動作が起こります．

(1) 次のThumb命令のアドレスを`r14_svc`に保存する

(2) `CPSR`を`SPSR_svc`に保存する

(3) プロセッサがIRQをディセーブル，Thumbビットをクリアし，`CPSR`内の関連ビットを変更することによってスーパバイザ・モードに入る

(4) PCを強制的に0x08番地に設定

次にARM命令の`SWI`ハンドラに入ります．通常の復帰命令でThumb実行状態が復元されます．

<<書式>>

```
        SWI     <8-bit immediate>
```

<<同等のARM命令>>

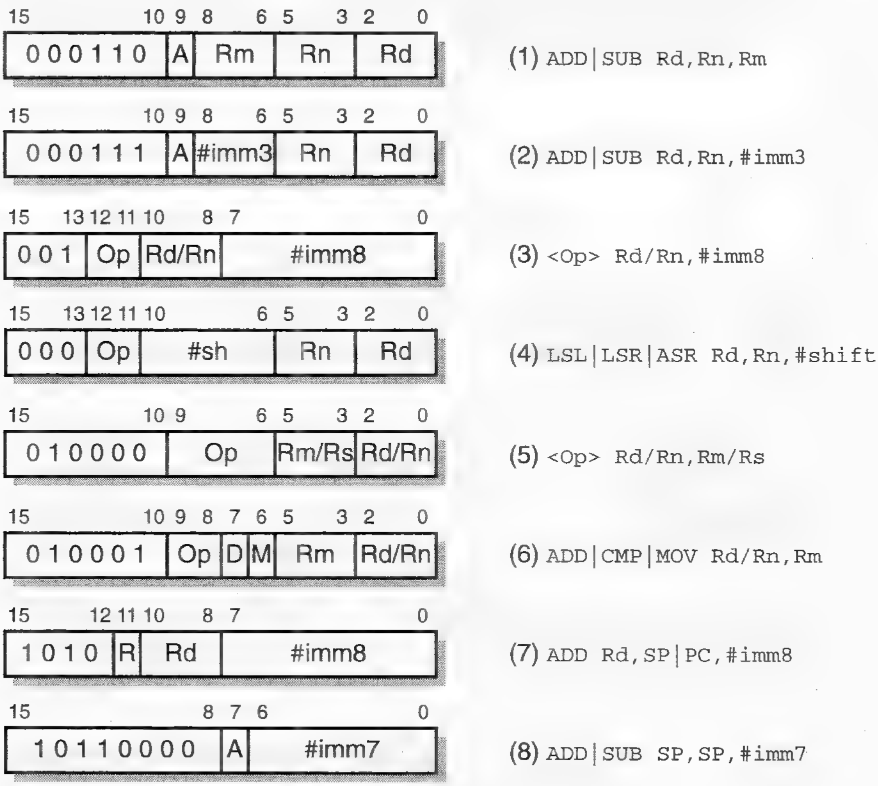
同等のARM命令はまったく同じアセンブラ構文規則をもちます．つまり，8ビット・イミディエートはARM命令の24ビット・フィールドを埋めるためにゼロ拡張されます．

これによってThumbコードで使用可能なSWIは，1600万あるARM SWIのうち最初の256だけに制限されることになります．

7.5 Thumb(サム)データ処理命令

Thumbデータ処理命令は，コンパイラがもっとも一般的に要求する処理を扱う複雑な形式を，高度に最適化したもので成り立っています．

これらの命令の機能は，十分明確です．含むものと除外するものの選択はそれほど明確ではありませんが，一般的なアプリケーションプログラムの必要性について詳しい理解の上に成り立っています．



〔図7.4〕Thumbデータ処理命令のバイナリ・エンコーディング

<<概要>>

すべての命令はARMデータ処理(乗算を含む)命令のどれかに対応します．ARMでは，一命令中でALU操作と一オペランドについての汎用シフト操作を同時にサポートしますが，Thumb命令セットはシフトとALU演算を別々の命令に分離します．したがって，シフト操作は，オペランド修飾子ではなくOPコードとして表れます．

<<書式>>

次のようにさまざまな命令形式があります．

```
1:     <op>    Rd, Rn, Rm          ; <op> = ADD|SUB
2:     <op>    Rd, Rn, #<#imm3>    ; <op> = ADD|SUB
3:     <op>    Rd|Rn, #<#imm8>     ; <op> = ADD|SUB|MOV|CMP
4:     <op>    Rd, Rn, #<#sh>      ; <op> = LSL|LSR|ASR
5:     <op>    Rd|Rn, Rm|Rs        ; <op> = MVN|CMP|CMN|..
     ; ..TST|ADC|SBC|NEG|MUL|LSL|LSR|ASR|ROR|AND|EOR|ORR|BIC
6:     <op>    Rd|Rn, Rm           ; <op> = ADD|CMP|MOV
     ;                                       (Hi regs)
7:     ADD     Rd, SP|PC, #<#imm8>
8:     <op>    SP, SP, #<#imm7>    ; <op> = ADD|SUB
```

<<同等のARM命令>>

Thumb命令セットに同等なものをもつARMデータ処理命令を下に列記しました．コメント・フィールド(右)にあるのがThumbです．

「Lo」を使う命令は，汎用レジスタ(r0からr7)を使います．

```
; ARM instruction                 Thumb instruction

  MOVS    Rd, #<#imm8>          ; MOV   Rd, #<#imm8>
  MVNS    Rd, Rm                ; MVN   Rd, Rm
  CMP     Rn, #<#imm8>          ; CMP   Rn, #<#imm8>
  CMP     Rn, Rm                ; CMP   Rn, Rm
  CMN     Rn, Rm                ; CMN   Rn, Rm
  TST     Rn, Rm                ; TST   Rn, Rm
  ADDS    Rd, Rn, #<#imm3>      ; ADD   Rd, Rn, #<#imm3>
  ADDS    Rd, Rd, #<#imm8>      ; ADD   Rd, #<#imm8>
  ADDS    Rd, Rn, Rm            ; ADD   Rd, Rn, Rm
  ADCS    Rd, Rd, Rm            ; ADC   Rd, Rm
  SUBS    Rd, Rn, #<#imm3>      ; SUB   Rd, Rn, #<#imm3>
  SUBS    Rd, Rd, #<#imm8>      ; SUB   Rd, #<#imm8>
  SUBS    Rd, Rn, Rm            ; SUB   Rd, Rn, Rm
  SBCS    Rd, Rd, Rm            ; SBC   Rd, Rm
  RSBS    Rd, Rn, #0            ; NEG   Rd, Rn
  MOVS    Rd, Rm, LSL #<#sh>    ; LSL   Rd, Rm, #<#sh>

; ARM instruction                 Thumb instruction

  MOVS    Rd, Rd, LSL Rs        ; LSL   Rd, Rs
  MOVS    Rd, Rm, LSR #<#sh>    ; LSR   Rd, Rm, #<#sh>
  MOVS    Rd, Rd, LSR Rs        ; LSR   Rd, Rs
  MOVS    Rd, Rm, ASR #<#sh>    ; ASR   Rd, Rm, #<#sh>
  MOVS    Rd, Rd, ASR Rs        ; ASR   Rd, Rs
  MOVS    Rd, Rd, ROR Rs        ; ROR   Rd, Rs
  ANDS    Rd, Rd, Rm            ; AND   Rd, Rm
  EORS    Rd, Rd, Rm            ; EOR   Rd, Rm
  ORRS    Rd, Rd, Rm            ; ORR   Rd, Rm
  BICS    Rd, Rd, Rm            ; BIC   Rd, Rm
  MULS    Rd, Rm, Rd            ; MUL   Rd, Rm
```

ときには「Lo」レジスタとも組み合わせ，「Hi」レジスタ(r8からr15)に処理を加えたり，ともに動作したりする命令は次のとおりです．

```
        ; ARM instruction                Thumb instruction

          ADD     Rd, Rd, Rm          ; ADD   Rd, Rm (1/2 Hi regs)
          CMP     Rn, Rm              ; CMP   Rn, Rm (1/2 Hi regs)
          MOV     Rd, Rm              ; MOV   Rd, Rm (1/2 Hi regs)
          ADD     Rd, PC, #<#imm8>    ; ADD   Rd, PC, #<#imm8>
          ADD     Rd, SP, #<#imm8>    ; ADD   Rd, SP, #<#imm8>
          ADD     SP, SP, #<#imm7>    ; ADD   SP, SP, #<#imm7>
          SUB     SP, SP, #<#imm7>    ; SUB   SP, SP, #<#imm7>
```

<<注意>>

(1) 「Lo」レジスタに処理を加えたりともに動作したりするデータ処理命令はすべて，条件コード・ビット(Sビットは ARM命令と同等にセットされている)を更新する．

(2) 「Hi」レジスタに処理を加えたり，ともに動作したりする命令は，条件コード・ビットを変更しない．例外はCMPで，これは条件コードのみを変更する．

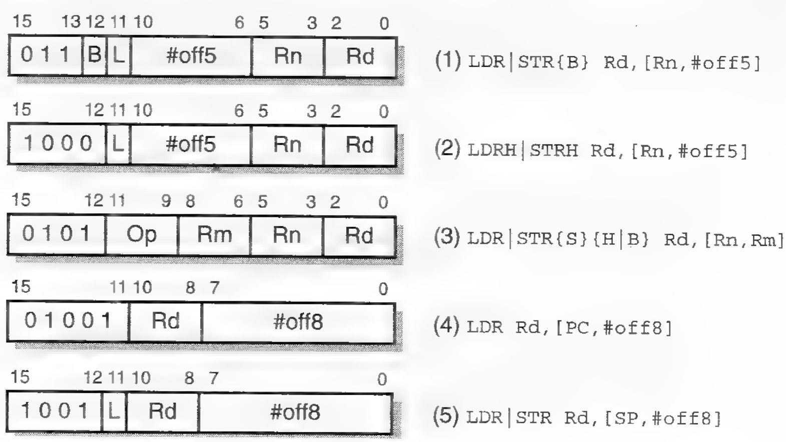
(3) 上で1または2の「Hi」レジスタを要求する命令は，片方または両方のレジスタ・オペランドが「Hi」レジスタ領域に指定されていなければならない．

(4) #imm3，#imm7，#imm8は，それぞれ3，7，8ビット・イミディエート・フィールドを表し，#shは5ビット・シフト量フィールドを表す．

7.6 Thumb(サム)単一レジスタ・データ転送命令

ここで，Thumb命令セットにあるARM命令の選別は複雑に見えますが，コンパイラが何を頻繁に行うかに基づいています．

リテラル・プール(PC相対)やスタック(SP相対)へアクセスのためのより大きなオフセットと，符号なしオペランド(ベースとオフセットまたはレジスタ)に比べ限られた符号付きオペランド(ベースとレジスタ・アドレシングのみ)のサポートに注目してください．



〔図7.5〕
Thumb単一レジスタ・データ転送のバイナリ・エンコーディング

<<概要>>

これらの命令は，ARM単一レジスタ転送命令から注意深く選び出したサブセットで，対応するARM命令とまったく同じセマンティクスをもっています．

すべての場合において，オフセットはデータ・タイプのサイズにスケーリングされますから，5ビット・オフセットの範囲は，バイトのロードおよびストア命令では32バイト，ハーフ・ワードのロードおよびストア命令では64バイト，ワードのロードおよびストア命令では128バイトです．

<<書式>>

次のように，さまざまな命令形式があります．

```
1:      <op>    Rd, [Rn, #<#off5>] ; <op> = LDR|LDRB|STR|STRB
2:      <op>    Rd, [Rn, #<#off5>] ; <op> = LDRH|STRH
3:      <op>    Rd, [Rn, Rm]       ; <op> = ..
                ; .. LDR|LDRH|LDRSH|LDRB|LDRSB|STR|STRH|STRB
4:      LDR     Rd, [PC, #<#off8>]
5:      <op>    Rd, [SP, #<#off8>] ; <op> = LDR|STR
```

<<同等のARM命令>>

これらのThumb命令に対応するARM命令とまったく同じアセンブラ書式をもちます．

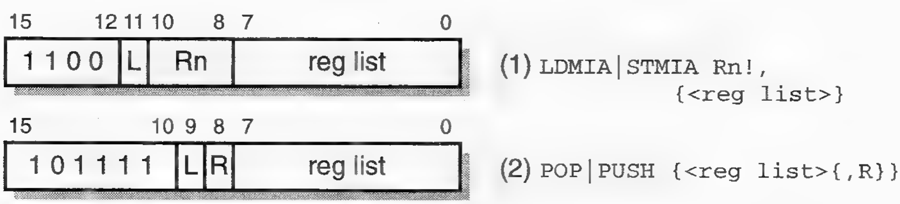
<<注意>>

(1) `#off5`，`#off8`は，5，8ビット・イミディエート・オフセットをそれぞれ表す．つねにアセンブラ書式ではオフセットをバイトで指定する．命令バイナリの5，8ビット・オフセットは，データ・タイプ・サイズに対応する．

(2) ARM命令では，符号付きストアと符号なしストアは，まったく同じ効果があるため，符号付きバリアントはロード命令だけがサポートしている．

7.7　Thumb(サム)複数レジスタ・データ転送命令

ARM命令セットと同様，Thumb複数レジスタ転送命令は，プロシージャのエントリおよび復帰とメモリブロック・コピーにおいて有用です．しかし，エンコーディングがきついので，二つの用法は分割しなければなりませんし，アドレシング・モードは制限されます．それ以外は，同等のARM命令の性質を引き継いでいます．



〔図7.6〕Thumb複数レジスタ・データ転送バイナリ・エンコーディング

<<概要>>

ブロック・コピー形式の命令はLDMIAとSTMIAアドレシング・モードを使います(図3.2参照)．ベース(基底)レジスタは「Lo」レジスタ(r0からr7)のうちどれかであり，レジスタ・リストはこれらレジスタの任意のサブセットを含むことができますが，ライトバックが常に選択されているためベース・レジスタ自体を含むことはできません(リストにベース・レジスタを含む複数レジスタ・ロード/ストア命令でライトバックが選択された場合の結果は予測不能)．

スタック形式はSP(r13)をベース・レジスタとして用い，やはり常にライトバックを用います．スタック・モデルはフル下降型だけです．レジスタ・リストに指定され得る8レジスタに加え，リンク・レジスタ(LRすなわちr14)も「PUSH」命令に含めることができ，PC(r15)は「POP」形式に含めることができます．ARMコードと同様，それによりプロシージャ・エントリおよびイグジット・シーケンスが最適化されるのです．

<<書式>>

<reg list>とは，レジスタ・リストであり，レジスタの範囲はr0からr7です．

```
LDMIA   Rn!, {<reg list>}
STMIA   Rn!, {<reg list>}
POP     {<reg list>{, PC}}
PUSH    {<reg list>{, LR}}
```

<<同等のARM命令>>

最初の二つのケースでは，同等のARM命令は同じアセンブラ書式をもち，後の二つのケースではPOPとPUSHが適切なアドレシング・モードに入れ替わります．

```
Block copy:

        LDMIA   Rn!, {<reg list>}
        STMIA   Rn!, {<reg list>}
Pop:

        LDMFD   SP!, {<reg list>{, PC}}
Push:

        STMFD   SP!, {<reg list>{, LR}}
```

<<注意>>

(1) ベース・レジスタは，ワード・アラインしていなければならない．そうでない場合，最下位の2アドレス・ビットを無視するレジスタや，アラインメント例外を生成したりするシステムがある．

(2) これらの命令は，すべてベース・ライトバックを用いるため，ベース・レジスタはレジスタ・リストに含むべきではない．

(3) レジスタ・リストのエンコーディングでは各レジスタに1ビットが当てられる．ビット0はr0が転送されるかどうかを示し，ビット1はr1を制御する．RビットはPOPとPUSH命令においてPCとLRオプションを制御する．

(4) アーキテクチャversion5Tでのみ，ロードしたPCの最下位ビットがThumbビットを更新します．このためThumbまたはARMの呼び出し側に直接復帰することができます．

7.8 Thumb(サム)ブレークポイント命令

Thumbブレークポイント命令は，相当するARM命令とまったく同様に動作します．ブレークポイント命令は，ソフトウェア・デバッグ時に使用します．この命令によって通常の命令実行を中断し，適切なデバッグ手続きを開始するのです．

15 … 8	7 … 0
10111110	xxxxxxxx

〔図7.7〕Thumbブレークポイント命令のバイナリ・エンコーディング

<<概要>>

デバッグ・ハードウェア・ユニットが適切に設定されている場合，この命令によってプロセッサはプリフェッチ・アボートを起こします．

<<書式>>

```
BRK
```

◆相当するARM命令

相当するARM命令もまったく同じ書式をもちます．BRK命令はARMアーキテクチャversion5Tを実装したARMプロセッサでのみサポートされています．

7.9 Thumb(サム)の実装

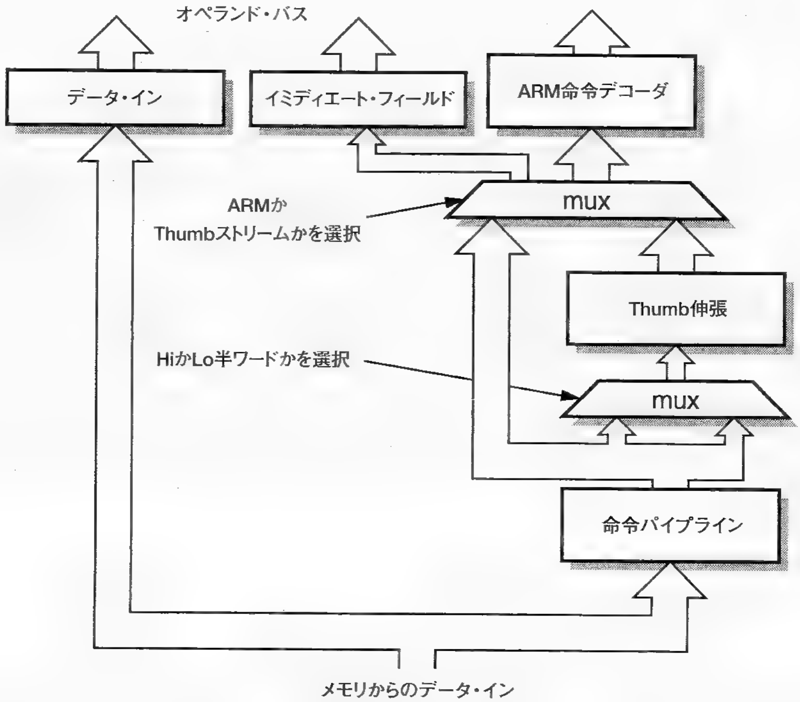
Thumb命令セットは，プロセッサ論理に比較的小さな変更を加えるだけで3段パイプラインARMプロセッサ・マクロセルに組み込むことができます(5段パイプラインに実装するほうが複雑)．最大の追加は，命令パイプラインのThumb命令伸張部で，この論理がThumb命令をARM命令に変換します．論理構成を図7.8に示します．

命令デコーダと連続する伸張論理の追加は，デコード遅延時間を増加するように思えますが，実際，ARM7パイプラインではデコード・サイクルのフェーズ1では，相対的に少ししか動作しません．したがって，伸張論理はサイクル時間に害を与えたりパイプライン遅延時間を増加することなく組み込まれています．そして，ARM7TDMI Thumbパイプラインは「3段パイプライン」でした説明のとおりに動作します．

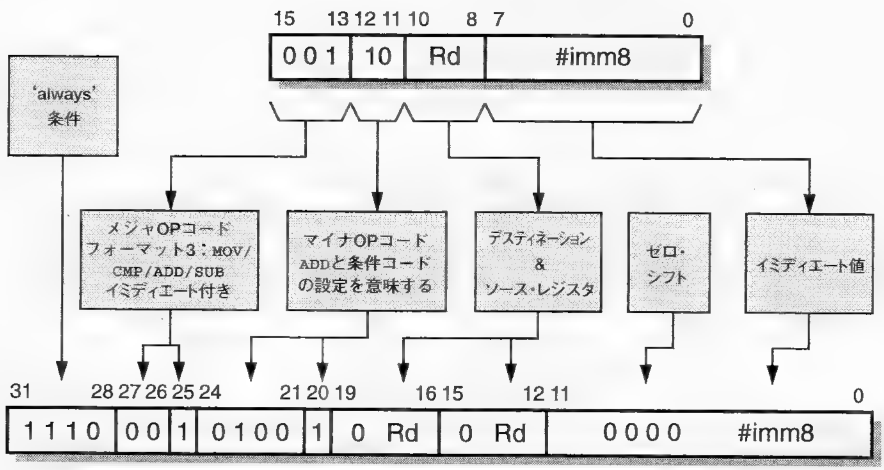
◆命令マッピング

Thumb伸張部は16ビットThumb命令を32ビットARM命令へ静的に変換します．ここに含まれるのは，メジャおよびマイナOPコードを変換するためのルックアップ，3ビット・レジスタ指定子をゼロ拡張して4ビット指定子にすること，必要に応じた他フィールドのマッピングです．

例として，Thumb「ADD Rd, #imm8」命令(7.5節を参照)の対応するARM「ADD Rd, Rd, #imm8」命令(5.7節を参照)へのマッピングを図7.9に示します．次のことに注目してください．



〔図7.8〕Thumb命令伸張部の構成



〔図7.9〕ThumbからARM命令へのマッピング

(1) Thumb命令の中で条件的なのは分岐だけなので，他のすべてのThumb命令を変換するのに「always」条件を使用する．

(2) Thumbデータ処理命令がCPSRの条件コードを修正すべきかどうかは，Thumb OPコード内に暗黙に指定されている．これはARM命令では明示しなければならない．

(3) Thumb命令の2アドレス形式は，つねにレジスタ指定子を複製することによってARM3アドレス形式にマップできる(一般的に逆は成立しない)．

伸張論理が単純であることは，Thumb命令セットの効率性にとって重要です．複雑で遅く，消費電力を食う伸張論理であれば，Thumbアーキテクチャを使うメリットがなくなってしまうでしょう．

7.10 Thumb(サム)アプリケーション

Thumbのどこにメリットがあるのかを理解するには，性質を調べる必要があります．Thumb命令は16ビット長でARM命令の機能性を半分のビット数にエンコードしています．しかし，Thumb命令はARM命令より文法的に意味する内容が少ないため，プログラムによってはARM命令より多くのThumb命令を必要とする場合があります．この比率はプログラムによって異なりますが，典型的な例ではThumbコードがARMコード空間の70%を要求することがあります．このように，Thumbソリューションを純粋なARMコード・ソリューションと比べると，次の特徴が明らかになります．

◆Thumbの性質

(1) ThumbコードはARMコード空間の70%を要求する
(2) ThumbコードはARMコードより使われる命令が40%多い
(3) 32ビット・メモリでは，ARMコードはThumbコードより40%速い
(4) 16ビット・メモリでは，ThumbコードはARMコードより45%速い
(5) ThumbコードはARMコードより使用する外部メモリ電力が30%少ない

したがって，性能重視なら，システムは32ビット・メモリを使ってARMコードを動作すべきです．コストと消費電力重視なら，16ビット・メモリ・システムとThumbコードがよいでしょう．しかし，両方の利点が得られる中間的な方法もあります．

◆Thumbシステム

(1) ハイエンド32ビットARMシステムでは，Thumbコードを一部の非クリティカル・ルーチンに使って電力やメモリを節約できる．

(2) ローエンド16ビット・システムでは，ARMコードで動作しているクリティカル・ルーチン用に少量のオンチップ32ビットRAMをもつことがあるが，すべての非クリティカル・ルーチンでオフチップThumbコードを使う．

上記の例のうち2番目が，Thumbが意図して開発されたアプリケーションに近いものです．携帯電話やポケットベル・アプリケーションはリアルタイム・ディジタル信号処理(DSP)機能を組み込んでおり，それがARMのフルパワーを要求することがありますが，それらの機能は少量のオンチップ・メモリにおさまるほど緊密にコードしたルーチンです．ユーザ・インターフェース，バッテリ管理システムなどを制御する複雑で大きなコードは時間クリティカルではないので，Thumbコードの使用によってオフチップROMが8，16ビット・バスで性能を発揮し，コスト節約，バッテリ寿命の延長につながるのです．

7.11 例題と演習

<例題7.1> 3.4節の「Hello World」プログラムを書き直し，Thumb命令を使うようにしなさい．ARMとThumbのコード・サイズ比較はどのように行うのか．

```
        AREA     HelloW,CODE,READONLY
SWI_WriteC       EQU    &0        ; output character in r0
SWI_Exit         EQU    &11       ; finish program
        ENTRY                     ; code entry point
START   ADR      r1, TEXT         ; r1 -> "Hello World"
LOOP    LDRB     r0, [r1], #1     ; get the next byte
        CMP      r0, #0           ; check for text end
        SWINE    SWI_WriteC       ; if not end print ..
        BNE      LOOP             ;   .. and loop back
        SWI      SWI_Exit         ; end of execution
TEXT    =        "Hello World",&0a,&0d,0
        END                       ; end of program source
```

これらの命令のほとんどが直接対応するThumb命令をもっていますが，ないものもあります．ロード・バイト命令は自動インデクスをサポートせず，スーパバイザ・コールは条件的に実行できません．したがって，Thumbコードでは多少修正が必要です．

```
        AREA     HelloW_Thumb,CODE,READONLY
SWI_WriteC       EQU    &0        ; output character in r0
SWI_Exit         EQU    &11       ; finish program
        ENTRY                     ; code entry point
        CODE32                    ; enter in ARM state
        ADR      r0, START+1      ; get Thumb entry address
        BX       r0               ; enter Thumb area
        CODE16                    ; Thumb code follows..
START   ADR      r1, TEXT         ; r1 -> "Hello World"
LOOP    LDRB     r0, [r1]         ; get the next byte
        ADD      r1, r1, #1       ; increment pointer    **T
        CMP      r0, #0           ; check for text end
        BEQ      DONE             ; finished?            **T
        SWI      SWI_WriteC       ; if not end print ..
        B        LOOP             ;   .. and loop back
DONE    SWI      SWI_Exit         ; end of execution
        ALIGN                     ; to ensure ADR works
TEXT    DATA
        =        "Hello World",&0a,&0d,&00
        END
```

上記リストで，Thumb命令セットにない機能を補うのに必要な追加命令は「**T」でマークしています．ARMコード・サイズは6命令とデータ14バイト，総計38バイトです．Thumbコード・サイズは8命令とデータ

14バイト(プロセッサがThumb命令を実行するようスイッチするのに必要なプリアンブルは無視する)，総計30バイトです．

この例は，Thumbコードを書く際に念頭に置かなければならない重要な点を多く示しています．

(1) アセンブラは，いつARMコードを生成し，いつThumbコードを生成するかを知らなければならない．「CODE32」，「CODE16」指示子が，その情報を伝える(これらはアセンブラに対する命令であり，それ自体は何のコードも生成しない)．

(2) プロセッサはコードを呼ぶときARM命令を実行しているため，Thumb命令を実行するように指示するには明示的な準備をする必要がある．r0が適切に初期化されていれば「BX r0」命令がこの役割を果たす．とくに重要なのは，最下位ビットr0が，分岐先でプロセッサがThumb命令を実行するためにセットされることである．

(3) Thumbコード「ADR」は，ワード・アラインしたアドレスしか生成できない．Thumb命令はハーフ・ワードなので，任意の数のThumb命令の後にきた位置がワード・アラインされているという保証はない．そのため，例題プログラムではテスト文字列の前に明示的な「ALIGN」がある．

ARMソフトウェア開発ツール・キット上で，このプログラムをアセンブル，動作するためには，Thumbコードを生成できるアセンブラを起動し，ARMulatorが「Thumb対応」プロセッサ・コアをエミュレートしなければなりません．プロジェクトマネジャーのデフォルト設定はARM6コアをターゲットにしており，32ビットARMコードしか生成しません．これはコード生成前に，プロジェクト・マネージャの「オプション」メニューから「プロジェクト」を選択し，「ツール」ダイアログ・ボックスで「TCC/TASM」を選択することで変更できます．これにより「ターゲット・プロセッサ」が自動的にThumb対応ARM7tにスイッチします．

そのほかは，ThumbコードはARMコードと同様にアセンブル，動作できます．

演習7.1.1 3.4および3.5節にある他のプログラムをThumbコードに変換し，元のARMコードとサイズを比較しなさい．

演習7.1.2 TCCを使ってCソース・プログラムからThumbコードを生成しなさい(通常どおり「Hello World」プログラムから始める)．Cコンパイラが生成したアセンブリ・コードを見なさい(「_s」オプションを使う)．ARMコードにコンパイルした同じプログラムとコード・サイズや実行時間を比較しなさい．

第8章 システム開発のためのアーキテクチャ・サポート

コンピュータ・システムの設計は，複雑な仕事です．組み込み型SOC(system-on-chip)となると，とくに難解です．開発が完全にCAD環境だけで行われるにもかかわらず，ファースト・シリコンにおいて，動作するだけでなく，製品としての必要な性能をすべて備えていなくてはなりません．チップができた後で設計上の欠点を直す道は，ソフトウェアしかありません．エラーを修正するためにチップに変更を加えるのは時間がかかりすぎるうえ，非常にコストがかかるのが普通です．

従来，マイクロプロセッサ・システム開発では，おもにICE(In-Circuit-Emulator)を使用していました．システム自体がマイクロプロセッサ・チップや多様なメモリ，周辺装置を組み込んだプリント回路基板でした．ICEを使うには，マイクロプロセッサをソケットから外し，代わりにICE装置とは，へその緒のようなケーブル付きのヘッダ・プラグでつないでいました．そしてICEがマイクロプロセッサの機能をエミュレートし，開発者にシステムの内部状態を見せ，プロセッサのレジスタやメモリ値の読み出し/変更/ブレークポイントの設定などを可能にしていました．

しかし，いまではマイクロプロセッサ自体が大きなチップの内部の一つのセルになったのですから，上記の構図は崩れました．チップの一部を取り外すなど，不可能です．いまのところ，ICEのすべての機能に代わる方法はありませんが，補助的なテクニックはいくつかあり，中にはプロセッサ・アーキテクチャ内の明示的なサポートを必要とするものもあります．本章では，ARMコア・ベースのシステム・チップ開発をサポートするテクニックと，コア内に組み込まれたアーキテクチャ上の特徴について説明します．

8.1 ARMメモリ・インターフェース

本節では，標準的なメモリ・パーツで構成されたメモリ・システムにARMプロセッサを接続するうえでの一般的な原則について述べます．メモリ・インターフェースの効率はシステム性能の重要な決定要素ですから，高性能システムを開発しようとする設計者はここで述べる原則をよく理解する必要があります．

最近のARMコアには，AMBAインターフェースがありますが(8.2節参照)，ここで説明する基本的な事柄の多くはやはり当てはまります．

◆ARMバス信号

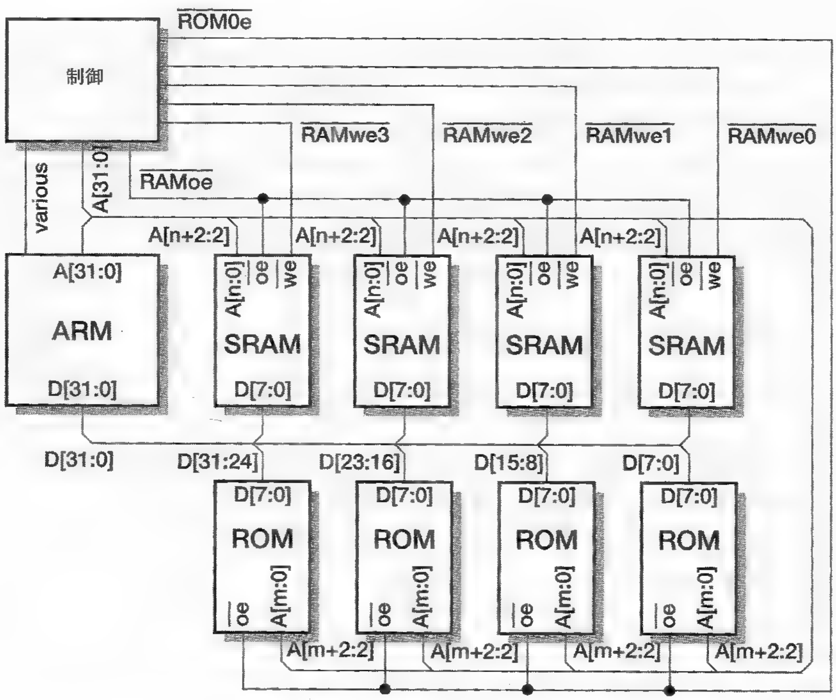
ARMプロセッサ・チップは，細部において異なるバス・インターフェースをもっていますが，本質的には似ています．メモリ・バス・インターフェース信号には以下があります．

(1) 32ビット・アドレス・バスである*A[31:0]*．アクセスするデータのバイト・アドレスを示します．
(2) 32ビット双方向データ・バスである*D[31:0]*．これでデータを転送します．
(3) メモリが必要とされているか($\overline{mreq}$)，アドレスがシーケンシャルか(*seq*)を指定する信号．これらの信号は，メモリ制御論理が適切に準備できるよう先行するサイクルで発行されます．
(4) 転送の方向($\overline{r}/w$)とサイズ(初期のプロセッサでは$\overline{b}/w$，新しいプロセッサでは*mas[1:0]*)を指定する信号．
(5) バス・タイミングおよび制御信号(*abe*，*ale*，*ape*，*dbe*，*lock*，*bl[3:0]*)．

◆単純なメモリ・インターフェース

もっとも単純な形のメモリ・インターフェースは，ROMとスタティックRAM(SRAM)を使った動作に適しています．このようなデバイスではサイクルの終わりまでアドレスが安定していなければなりません．このためには，新しいプロセッサではアドレス・パイプラインをディセーブル(*ape*を“L”に固定)にし，旧いプロセッサではアドレス・バスをリタイミング(*ale*を$\overline{mclk}$に接続)します．すると図8.1のようにアドレス・バスとデータ・バスをメモリ部品に直接，接続することができます．ここには出力イネーブル信号($\overline{RAMoe}$と$\overline{ROMoe}$)および書き込みイネーブル($\overline{RAMwe}$)も示されています．

この図は，SRAMとROMの標準的な構成である8ビット・メモリ部品の接続を示しています．32ビット・メモリを形成するには各タイプ4個ずつ必要であり，各デバイスはバスの1バイト・セクションに接続しています．図中で，デバイスのバス番号はデバイス内に示し，それが接続するバス・ワイヤはデバイスの外に示していますから，たとえばARMの最も近くにあるSRAMは，ピン*D[7:0]*がバス・ピン*D[31:24]*に接続し，それがARMのピン*D[31:24]*に接続しています．



〔図8.1〕基本ARMメモリ・システム

最下位2本のアドレス・ラインである*A[1:0]*は，バイト選択に使用するため，制御論理が使用し，メモリには接続していません．したがってメモリ・デバイス上のアドレス・ラインは*A[2]*以上に接続しています．使用する正確な番号は，メモリ・パーツのサイズによって決まります（たとえば，128KバイトのROMパーツは*A[18:2]*を使用）．

ARMはバイトとワードの両方の読み出しを実行しますが，メモリ・システムはその違いを無視して（ただし，多少の電力が無駄になる）つねにワードを出します．ARMは，アドレス指定されたバイトだけを抜き出し，ワードの残り部分は無視します．ですからROMは個別のイネーブルが不要であり，16ビット・デバイスを使用しても問題はありません．しかし，バイト書き込みの場合は個別のバイト・イネーブルが必要ですから，制御論理が4バイトの書き込みイネーブル制御を生成する必要があります．そのため，RAMが個別のバイト・イネーブルを組み込んでいないかぎり，幅広いRAMを使用するのは困難（かつ非効率的）です．なぜなら個別のバイトを書き込むには，リード・モディファイ・ライト・メモリ動作が必要になるからです．多くのプロセッサは，バイト書き込みのサポートを必要としていますから，もし本当にバイトより幅の広いデータをもつRAMが出回るようになれば，おそらく個別バイト・イネーブルを組み込むことでしょう．

◆制御論理

制御論理は，以下の機能を果たします．

(1) いつRAMをアクティブにするか，いつROMをアクティブにするかを決定する

この論理はシステムのメモリ・マップを決定します．プロセッサはリセット後，ゼロ番地から起動しますから，RAMは初期化されてなく，ROMを見つける必要があります．したがって最も単純なメモリ・マップでは，*A[31]*が“L”のときROMをイネーブルし，“H”のときRAMをイネーブルします（ほとんどのARMシステムでは，起動の直後にメモリ・マップを変更してメモリの最下位にRAMを置き，例外ベクタが変更できるようにする）．

(2) 書き込み動作中にバイト書き込みイネーブルを制御する

ワード書き込みの間は全バイト・イネーブルがアクティブでなければならず，バイト書き込みの間はアドレス指定されたバイトだけがアクティブでなければなりません．ARMがハーフ・ワードをサポートしている場合は，ハーフ・ワード書き込みの間，四つのイネーブルのうち二つがアクティブでなければなりません．

(3) プロセッサが継続する前にデータの準備ができていることを確認する

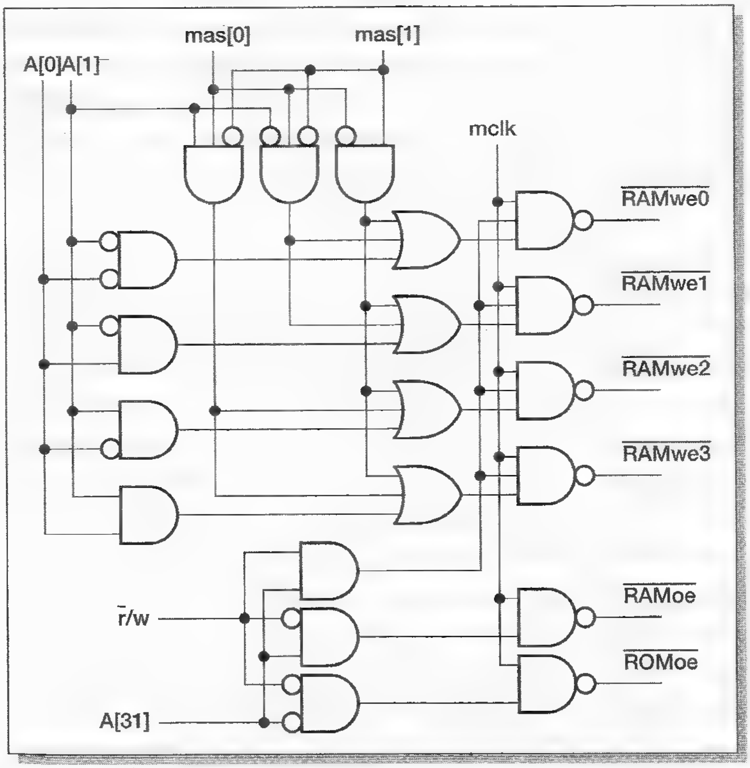
もっとも簡単な方法は，$\overline{mclk}$をゆっくりと動作させることによって，全メモリ・デバイスが1クロック・サイクルの間に確実にアクセスできるようにすることです．もっと高度なシステムでは，RAMアクセスに合わせてクロックを設定し，（通常，低速の）ROMやペリフェラルにアクセスする場合はウェイト状態を使用することもあります．

上記の機能を果たすために必要な論理はきわめて単純で，図8.2に示すとおりです（論理のすべてが一つのプログラマブル論理デバイスで実装できる）．おそらく設計が難しいのは，双方向データ・バスに関する部分です．ここで非常に重要なのは，つねに一つのデバイスだけがバスを駆動するようにすることです．したがって書き込みサイクルでバスを反対向きにしたり，ROMからの読み出しとRAMからの読み出しを切り替えたりする場合には注意してください．図に示したソリューションでは，$\overline{mclk}$が“H”のときに適切なデータ・ソースをアクティブにし，$\overline{mclk}$が“L”のときに全ソースをオフにします．そのためプロセッサのデータ・バス・イネーブルである*dbe*も*mclk*に連動します．これは非常に保守的な方法で，使用可能な最大クロック周波数を制限するためシステム性能を妥協しなければならないこともよくあります．

この設計では，ARM出力がクロック・サイクルの終わりまで安定していることを前提にしています．これは，新しいプロセッサでは，アドレス・パイプライン・イネーブル（ape）制御入力が“L”に固定されている場

合に相当します．旧いプロセッサの場合は，*ale*=$\overline{mclk}$を使ってアドレス出力をリタイムしなければなりませんが，$\overline{r}$/*w*と$\overline{b}$/*w*（*mas[1]*を置き換え，*mas[0]*は“L”に接続）をリタイムするには$\overline{mclk}$が“L”のときに開く外部のトランスペアレント・ラッチが必要になります．

この単純なメモリ・システムでは，$\overline{mreq}$（または*seq*）を使用していません．毎サイクルでメモリをアクティブにしているだけです．ARMは純粋なメモリ・アクセスにおいてしか書き込みサイクルを要求しませんから，これが安全なのです．すべての内部およびコプロセッサ・レジスタ転送サイクルの間，$\overline{r}$/*w*制御は“L”のままになります．



〔図8.2〕簡単なARMメモリ・システムの制御論理

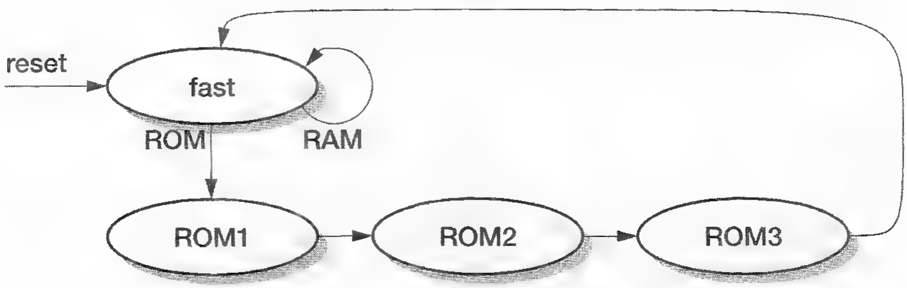
◆ウェイト・ステート

このシステムでクロックを速くしようとすると，もっとも遅いパスが追いつけなくなるときに動作が停止してしまいます．これは通常，ROMアクセスです．RAMアクセスに合わせてクロックを調整でき，ROMにもっとアクセス時間を与えるためにウェイト・サイクルを導入できれば，システム性能をさらに向上することができます．通常，ROMはアクセス当たり決まったクロック・サイクル数を与えられており，その正確な数字はクロック速度とROMデータ・シートによって決まります．ここでは，アクセス時間を4サイクルとします．

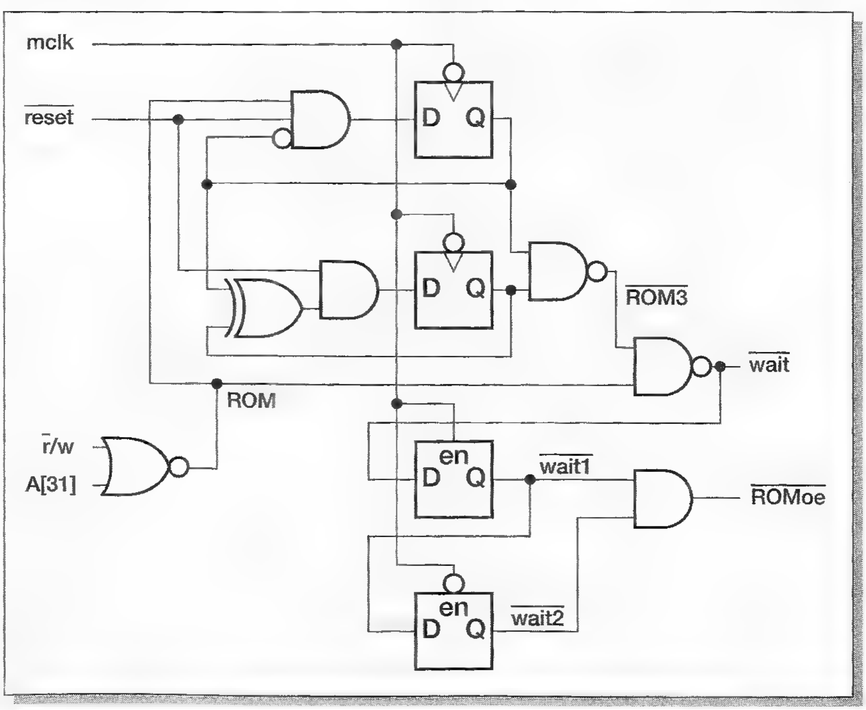
メモリ制御論理は，単純な有限ステートマシンを組み込んでROMアクセスを制御する必要があります．適切な状態遷移図は図8.3のとおりです．ARMの$\overline{wait}$入力をアサートすることによってROMアクセスを4サイクルに引き伸ばすには，三つのROMステートを使用します．設計上の問題となるのは，アドレスが現在のサイ

クルの早期に有効になるようリタイムされており，$\overline{wait}$は$\overline{mclk}$の立ち上がり前にアサートしなければならない点です．ステートマシン$\overline{wait}$を生成するクロック・エッジはありませんから，$\overline{wait}$は単純なステートマシン出力としては生成できないことになります．もう一つの問題は，延長された$\overline{ROMoe}$をグリッチなしに生成することです．

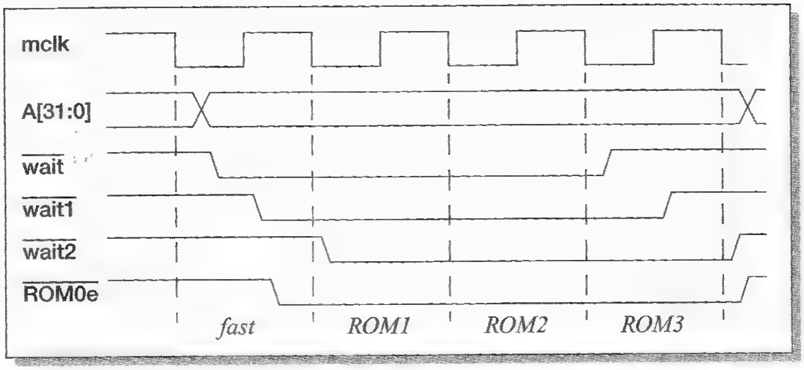
図8.4に一つの例を示しています．ステートマシンは，二つのエッジ・トリガ型フリップフロップを使用する同期カウンタです．重要なのはROM3ステートだけで，ROMアクセスを検知するたびにアクティブになってしまう$\overline{wait}$をゲートします．$\overline{wait}$を起点に延長したクリーンな$\overline{ROMoe}$を生成するには，二つのレベル・センシティブ・ラッチを使用します．この回路のタイミング・ダイアグラムは図8.5のとおりです．これで論理の動作がはっきりわかります．



〔図8.3〕ROMのウェイト制御状態遷移図



〔図8.4〕ROMのウェイト・ステート生成回路

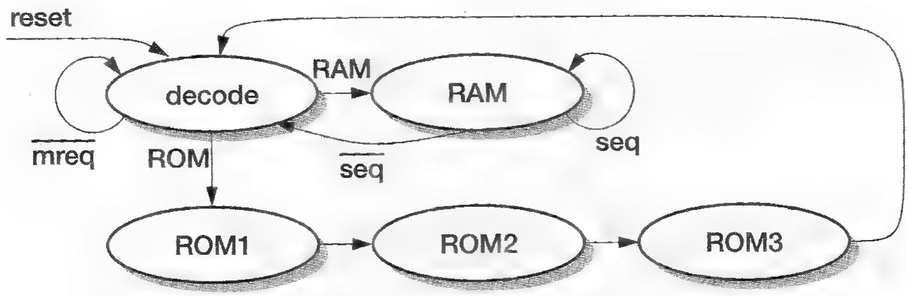


〔図8.5〕ROMのウェイト・ステート論理のタイミング図

◆シーケンシャル・アクセス

システムの動作がもっと速くなると，1クロック・サイクルで新しいアドレスをデコードし，RAMアクセスを実行することが困難になります．そこで，未知のアドレスが発行されるとつねに追加サイクルを挿入し，アドレス・デコードの時間を作ることになります．未知でないアドレスはシーケンシャル・アドレスだけですが，通常のプログラムではこれが全アドレスの75%を占めます．

ここで，メモリを使用しないサイクルについても認識し始めなければなりません．そのようなサイクルがフル・クロック速度で動作してはならないという理由は何もないからです．該当する状態遷移図は図8.6のとおりです．



〔図8.6〕アドレス・デコードのためのウェイトのある状態遷移図

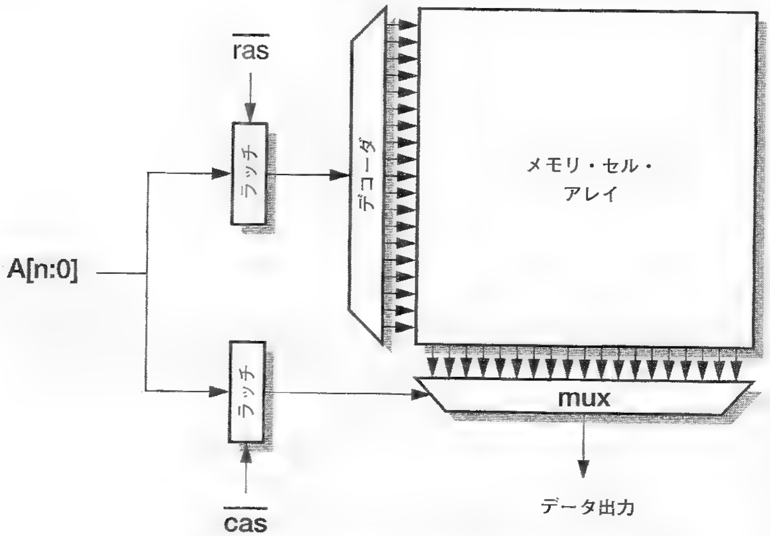
◆DRAM

もっとも安価なメモリ・テクノロジ(ビット当たりの単価の点で)は，ダイナミック・ランダム・アクセス・メモリ(DRAM)です．「ダイナミック」なメモリでは，情報がコンデンサに電荷として保存され，徐々に漏れ出します(msecぐらいの間に)．このメモリ・データは漏れ切ってしまう前に読み出して書き直さなければ(リフレッシュする)なりません．メモリをリフレッシュする役割を果たすのは通常，メモリ制御論理であり，プロセッサではありませんから，ここで触れなければならない問題ではありません．問題は，図8.7に示したメモリの内部構成です．

ほとんどのメモリ・デバイスと同様，DRAMのストレージ・セルはだいたい正方形のマトリックス状に並

んでいます．ほとんどのメモリ・デバイスと違うところは，この構成がユーザから見えることです．マトリックスは行（ロウ）と列（カラム）で表現されますが，DRAMでは行と列のアドレスを別々に同じマルチプレクスされたアドレス・バスで受け取ります．まず行アドレスが与えられ，“L”アクティブの行アドレス・ストローブ信号（$\overline{ras}$）を使ってラッチされます．次に列アドレスが与えられ，“L”アクティブの列アドレス・ストローブ信号（$\overline{cas}$）を使ってラッチされます．次のアクセスも同じ行にあれば，最初に新しい行アドレスを与えることなく，新しい列アドレスを与えることができます．$\overline{cas}$オンリ・アクセスはセル・マトリックスをアクティブにしないため，完全な$\overline{ras\text{-}cas}$アクセスより2，3倍速くデータを与えることができ，電力消費も大幅に少なくなります．そのため，可能であればつねに$\overline{cas}$オンリ・アクセスを使用するのが有利です．

難しいのは，メモリ・アクセスの早期に新しいアドレスが前のアドレスと同じ行にあることを検出することです．新しいアドレスの関連ビットを前のアドレスの対応ビットと比較するのが，ほとんどいつも間に合わないのです．



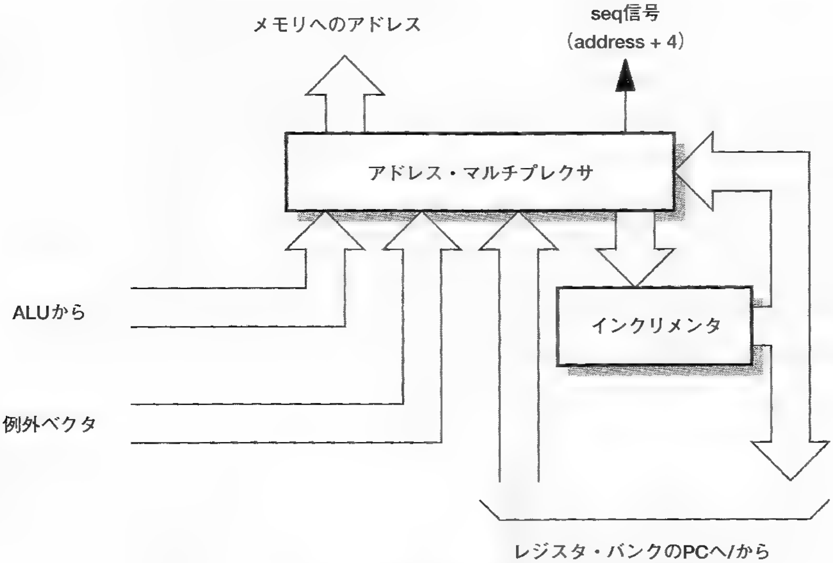
〔図8.7〕DRAMメモリ内部構造

◆ARMアドレス・インクリメンタ

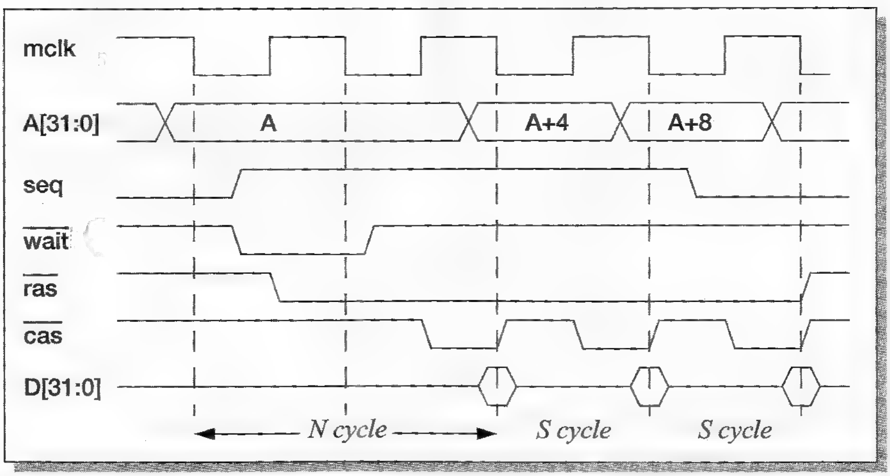
ARMが採用したソリューションは，ほとんどのアドレス（通常75%）がアドレス・インクリメンタで生成される事実を利用しています．ARMのアドレス選択論理（図8.8）は，次のサイクルのアドレスを四つのソースのうち一つから選択します．これらのソースの一つがインクリメンタです．次のアドレスがインクリメンタから来る場合，ARMは*seq*出力をアサートすることによって外の世界に明示します．すると外部論理が前のアドレスの行境界をチェックします．前のアドレスが行の終わりになく，*seq*信号がアサートされていれば，$\overline{cas}$オンリ・アクセスを実行することができます．

このメカニズムは，同じDRAM行に入るアクセスをすべて検知できるわけではありませんが，ほとんどを検知しますし，実装して利用するのが非常に簡単です．*seq*信号と前のアドレスは全部，問題となるサイクルの0.5クロック・サイクル前から有効で，メモリ制御論理には十分な時間があります．

典型的なDRAMタイミング・ダイアグラムは図8.9のとおりです．最初，行アドレスがストローブされ，ノン・シーケンシャル・アクセスに2クロック・サイクルかかりますが，後続のシーケンシャル・アドレスが$\overline{cas}$



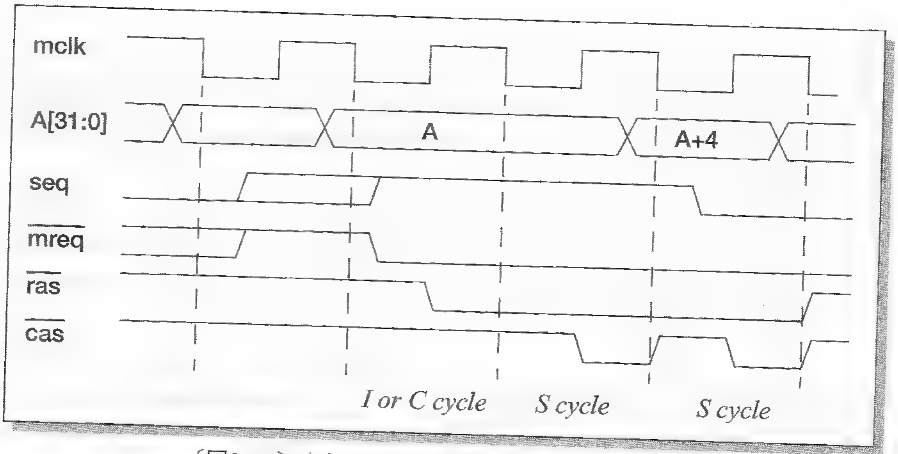
〔図8.8〕ARMアドレス・レジスタ構造



〔図8.9〕DRAMタイミング図

オンリ・アクセスを使って1クロック・サイクルで動作します(*ape*または*ale*を"H"にして，早期アドレス・タイミングが使われていることに注意).

*seq*信号のもう一つの使い道は，先行する内部またはコプロセッサ・レジスタ転送サイクルと同じアドレスを使用するサイクルであることを示すことであり，これもDRAMアクセス時間を向上するのに利用できます．*seq*が非常に早期に入手できた場合に限り，DRAMアクセスが前のサイクルで開始します．典型的なタイミングは図8.10のとおりです(このシーケンスの間，$\overline{wait}$はアクティブにならない).



〔図8.10〕内部サイクル後のDRAMタイミング図

◆ペリフェラル・アクセス

ほとんどのシステムは，ここまでに説明したメモリ部品に加えて周辺装置を組み込んでいます．周辺装置のアクセス速度は遅いのが普通ですが，上記でROMアクセスについて使ったのと同様のテクニックでインターフェースすることができます．

8.2 AMBA (Advanced Microcontroller Bus Architecture)

ARMプロセッサ・コアは高速キャッシュ・インターフェース向けに最適化したバス・インターフェースをもっています．キャッシュがあってもなくても，複雑なシステム・チップのコンポーネントとしてコアを使用する場合，ARMがほかのオンチップ・マクロセルと通信するために何らかのインターフェースが必要です．

このインターフェースを設計するのは難しくありませんが，考えうる方法は多くあります．毎回異なる方法を使うことは，設計リソースを消費し，周辺マクロセルの再利用を妨げることになります．無駄を避けるために，ARM社はAMBA (Advanced Microcontroller Bus Architecture) を定義して，異なるマクロセルのオンチップ接続を標準化しています．このバス・インターフェースに準拠して設計したマクロセルは将来のシステム・チップ向けキット部品と見なすことができ，最終的に既存マクロセルの新しい組み合わせを使って複雑なチップを設計することが簡単になります．

◆AMBAバス

AMBA仕様には三つのバスが定義されています．

(1) Advanced High-performance Bus (**AHB**)

高性能システム・モジュールを接続するのに使用します．バースト・モードのデータ転送と分割トランザクションをサポートし，全タイミングが単一クロック・エッジを参照しています．

(2) Advanced System Bus (**ASB**)

高性能システム・モジュールを接続するのに使用します．データおよびバースト・モードのデータ転送をサポートします．

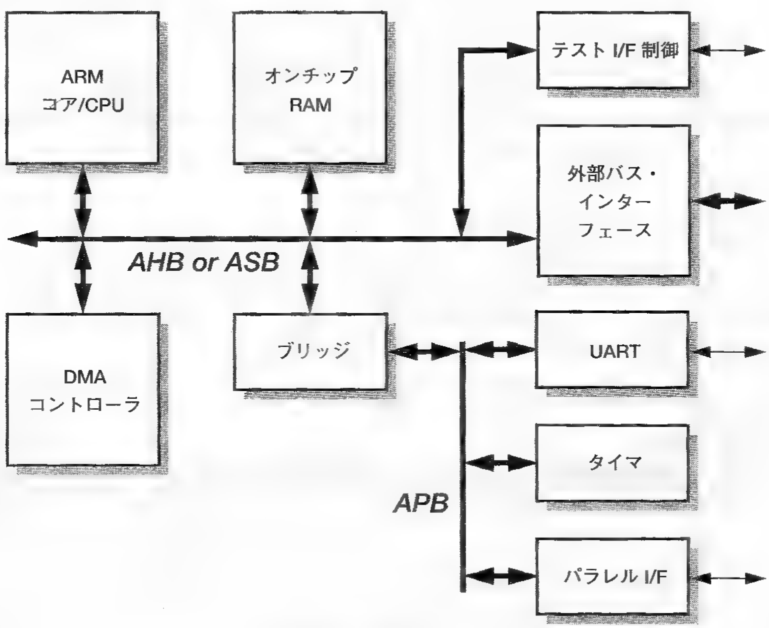
3) Advanced Peripheral Bus (**APB**)

低性能周辺用の単純なインターフェースとなります

一般的なAMBAベースのマイクロコントローラは，図8.11のようにAHBまたはASBとAPBを組み込んでいます．ASBは旧い形のシステム・バスであり，後でAHBが高性能，合成，タイミング検証に対するサポートを向上するために発表されました．

APBは通常，ローカルの2次バスで，AHBやASB上では一つのスレーブ・モジュールになります．

以下の節では，システム・バスをASBとして説明します．AHBについて詳しくは，節の終わりで述べます．



〔図8.11〕典型的なAMBAベース・システム

◆アービトレーション(調停)

バス・トランザクションは，中央アービタからアクセスを要求するバス・マスタが引き起こします．競合する要求があればアービタが優先順位を決め，その設計はシステム固有の問題です．ASBは従うべきプロトコルを以下のように指定するだけです．

(1) マスタxが中央アービタに要求(*AREQx*)する．

(2) バスが使用可能なとき，アービタは許可(*AGNTx*)をマスタに返す(アービトレーションは，どの許可を送出するか決める際にバス・ロック信号(*BLOCK*)を考慮に入れ，不可分なバス・トランザクションが妨害されないようにしなければならない)．

◆バス転送

マスタはバスへのアクセスが許可されると，アドレスと制御情報を発し，転送タイプと応答すべきスレーブ・デバイスを示します．トランザクション・タイミングを定義するために，次の信号を使用します．

(1) バス・クロック(*BCLK*)

通常はARMプロセッサクロックである$\overline{mclk}$と同じです．

許可されたバス・マスタは，次の信号を用いてバス・トランザクションに進みます．

(2) バス・トランザクション(*BTRAN[1:0]*)

次のバス・サイクルがアドレスのみ，シーケンシャル，非シーケンシャルのいずれになるか示します．許可信号によってイネーブルされ，参照するバス・サイクルより先行します．

(3) アドレス・バス(*BA[31:0]*)

(全アドレス空間を使う必要がない場合，すべてのアドレス線を接続する必要はありません．マルチプレクスされた実装ではアドレスがデータ・バスに送られます)．

(4) バス転送方向(*BWRITE*)

(5) バス保護信号(*BPROT[1:0]*)

命令，データいずれのフェッチか，スーパバイザ，ユーザいずれのアクセスかを示します．

(6) 転送サイズ(*BSIZE[1:0]*)

バイト，ハーフ・ワード，ワード，いずれの転送かを示します．

(7) バス・ロック(*BLOCK*)

マスタがバスを保持し，不可分なリード・モディファイ・ライト・トランザクションを完了するのを可能にします．

(8) データ・バス(*BD[31:0]*)

書き込みデータを送出し，読み出しデータを受け取るのに使用します．マルチプレクス化したアドレスとデータをもつ実装では，アドレスもこのバスで伝送されます．

スレーブ・ユニットは，要求されたトランザクションを直ちに処理することができます．つまり，*BD[31:0]*で書き込みデータを受け入れる，読み出しデータを送出する，もしくは次の応答のうちいずれかを送ります．

(9) バス待ち(*BWAIT*)

スレーブ・モジュールが現在のサイクルでトランザクションを完了できない場合，ウェイト・ステートを挿入します．

(10) バス・ラスト(*BLAST*)

スレーブがシーケンシャル・バーストを終了し，バス・マスタが新しいバス・トランザクション要求を発して先へ進むようにさせます．

(11) バス・エラー(*BERROR*)

完了できないトランザクションを表示します．マスタがプロセッサであれば，転送をアボートします．

◆バス・リセット

ASBは，多数の独立したオンチップ・モジュールをサポートしており，モジュールの多くはデータ・バス(と，いくつかの制御線)を駆動することができます．全モジュールがバス・プロトコルに従うとすると，つねに一つのモジュールがいずれかのバスを駆動していることになります．しかし，パワーオン直後は，すべてのモジュールが未知状態になります．クロック・オシレータが起動後に安定するには少し時間がかかりますから，全モジュールを既知状態にするような高い信頼性のクロックが利用できがないことがあります．

どのような場合でも，二つ以上のモジュールが起動し，バスを逆方向に駆動しようとすると，出力ドライブのクラッシュが電源にクロウ・バー(crow-bar)問題を起こし，チップがまったく正常に起動しなくなります．

正しくASBが起動するには，非同期リセット・モードにして，全ドライバをクロックから独立してバスか

ら外すようにします．

◆テスト・インターフェース

AMBAは，「テスト・インターフェース・コントローラ」を用いてモジュラ・テストをサポートすることもできます．これにより，外部テスタをASB上のバス・マスタにしてAMBA上の各モジュールを別々にテストすることができます．

テスト・モードをサポートするには，テスタがASBに32ビット双方向ポートでアクセスできることが条件です．外部メモリや周辺装置への32ビット双方向データ・バス・インターフェースがあればよいのです．オフチップ・データ・インターフェースが16または8ビット幅なら，テスト・アクセスにはアドレス線などの他信号を32本をもたなければなりません．

二つのテスト要求入力（*TREQA*と*TREQB*）で定義されたプロトコル，コントローラのアドレス・ラッチとインクリメンタを使用することにより，テスト・インターフェースでASBアドレス・バスおよびデータ・バスの制御が可能になります．適切に設計されたマクロセル・モジュールの場合，最高32ビットのグループを成すインターフェース信号すべてにアクセスできます．たとえば，ARM7には13ビットの制御入力およびコンフィグレーション入力，32ビット・データ入力，15ビット状態出力，32ビット・アドレス出力およびデータ出力があります．テスト・ベクタを適用し，有限オートマトンが定義するシーケンスにしたがって応答を検出します．有限オートマトンの状態遷移は*TREQA*と*TREQB*が制御しています．

AMBAマクロセル・テストは，本章前半の「マクロセル・テスト」で述べたJTAGに基づく方法と比較されることがあります．AMBAの方法はJTAGほど一般的ではないかもしれませんが，パラレル・テスタ・インターフェースのおかげでテスト・コストを低減します．

◆APB

ASBは，比較的高性能なオンチップ相互接続を提供し，プロセッサ，メモリ，周辺マクロセルに適合し，組み込みインターフェースを高度化します．非常に単純，低速の周辺装置では，インターフェースのオーバヘッドは大きすぎます．「APB（Advanced Peripheral Bus）」は単純なスタティック・バスで，ASB上ではスタブとして動作し，きわめて単純な周辺マクロセルに最小限のインターフェースを提供します．

選択信号がローカルのイネーブルとして動作するようなローカル・デコーディングが時間的な余裕を確保できるように，アドレスと制御信号はすべてタイミング・ストローブに対してセットアップとホールドが規定されています．APB転送は*PCLK*にタイミングを合わせてあり，全APBデバイスは*PRESETn*でリセットします．

このバスには，アドレス（*PADDR[n:0]*；32ビット全部は通常必要ない）と，接続される周辺が必要とするだけの幅の読み出しバスと書き込みバス（*PRDATA[m:0]*と*PWDATA[m:0]*，ここで*m*は7，5，31）と，読み出し/書き込み方向指示線（*PWRITE*）と，個々の周辺の選択ストローブ（*PSELx*）と，周辺のタイミング・ストローブ（*PENABLE*）が1本あります．したがってローカル・デコードの時間をもたせ，選択ストローブはローカル・イネーブルとして動作します．単純なレジスタ・マッピングに基づくスレーブ・デバイスである周辺装置は，最小限の論理オーバヘッドで直接インターフェースをとることができます．

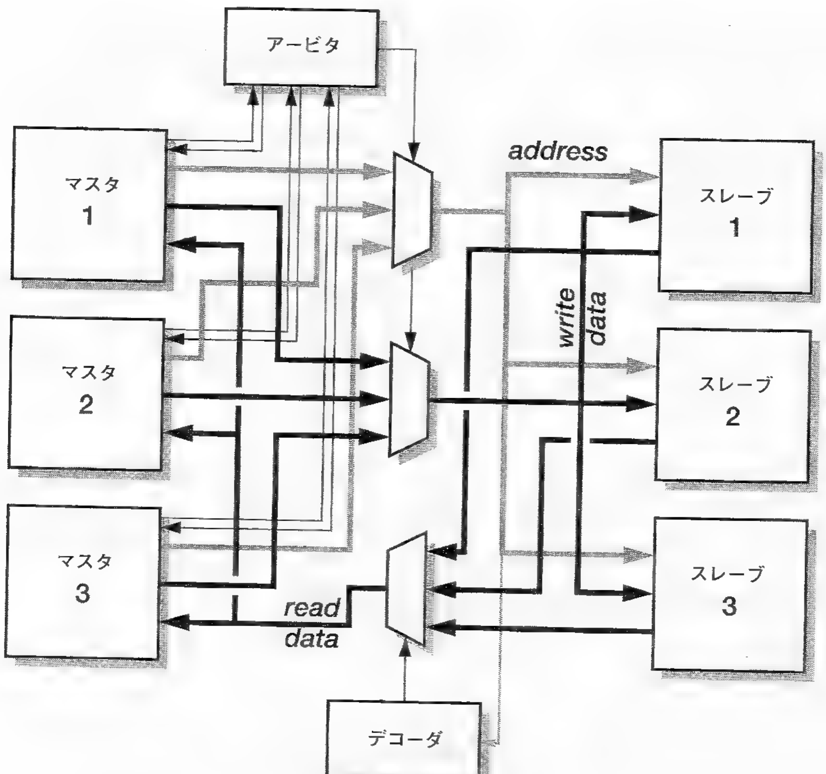
◆AHBバス

AHBは，ARM1020E（12.6節で述べる）ベースのシステムなど非常に高性能なシステムにおいてASBに取って代わるためのものです．

AHBは以下の点でASBと異なります．

- 分割トランザクションをサポートしているため，応答遅延時間の長いスレーブは，伝送するデータを準備する間，バスを開放することができます．
- 全動作を制御するのに一つのクロック・エッジを使用するため，合成や設計検証に便利です（スタティック・タイミング分析や類似のツールの使用により）．
- 3ステート・ドライバでの双方向バスではなく，中央マルチプレックス・バス方式を使用しています（図8.12参照）．
- 64または128ビットの幅広いデータ・バス構成をサポートします．

マルチプレックス・バス方式には，多くの配線が必要になるように思えますが，双方向バスは設計者に多くの問題をもたらしますし，合成システムにとってはもっと多くの問題があります．たとえば，チップの設計ルールが小さくなるにつれ，配線遅延が性能的に大きな問題になり始めますが，片方向バスの場合は双方向バスと違ってリピータ・ドライバを簡単に挿入できるという利点があります．



〔図8.12〕AHBマルチプレックス・バス方式

8.3 ARM参照ペリフェラル仕様

本章で説明してきたシステム開発は，テストや，プロセッサやシステム状態への低レベル・アクセスに主眼を置いたものです．AMBAは，一つのチップ上でハードウェア・コンポーネントを接続するシステマティッ

クな方法ですが，ソフトウェア開発についてはやはり新しいチップごとに最初の原則から出発しなければなりません．

システム開発者がより高いレベルから出発する，たとえばソフトウェアを特定のリアルタイムOS上に載せるところから出発しようとすると，基本的OS機能をサポートする多くのコンポーネントが必要になります．「ARM参照ペリフェラル仕様」はそのような基本的コンポーネント群を指定し，OSを動作しつつ特定アプリケーション向けシステム拡張の余地が十分ある枠組みをつくります．

参照ペリフェラル仕様の目的は，準拠している実装の間でソフトウェアの移植を簡単にし，新システム上でのソフトウェア開発の出発レベルを上げることです．

◆ベース・コンポーネント

参照ペリフェラル仕様は次のコンポーネントを指定しています．

(1) メモリ・マップ

割り込みコントローラ，カウンタ・タイマ，リセット・コントローラが，さまざまなベース・アドレスをとることを可能にし，かつ，これらのベース・アドレスからのレジスタ・オフセットを定義します．

(2) 割り込みコントローラ

送信および受信用通信チャネルのための定義済み割り込みメカニズムなど，定義済み機能群をもちます．

(3) カウンタ・タイマ

多数の定義済み機能をもちます．

(4) リセット・コントローラ

定義済みブート動作，パワーオン・リセット検出，「割り込み待ち」一時停止モード，識別レジスタをもちます．

これらのコンポーネントとともに使用するARMコアはとくに指定されません．なぜなら，システム・プログラマ・モデルには影響しないからです．

◆メモリ・マップ

システムは，割り込みコントローラ，カウンタ・タイマ，リセット一時停止コントローラのベース・アドレス(それぞれ，*ICBase*，*CTBase*，*RPCBase*)を定義しなければなりません．

これらのアドレスは参照ペリフェラル仕様には定義されていませんが，レジスタのアドレスはすべてこれらのベース・アドレスのいずれかに対して相対的に定義されています．

◆割り込みコントローラ

割り込みコントローラは，32までのレベル・センシティブな*IRQ*ソースと一つの*FIQ*ソースのステータスを調べたり，イネーブル，ディセーブルする一様な手順を提供します．各割り込みソースはソースをイネーブルするマスク・ビットをもちます．メモリ位置は*ICBase*からの固定オフセットで定義されて，マスクなし状態，マスク状態，マスク割り込み状態を検査したり割り込みソースをセット，クリアしたりします．

五つの*IRQ*ソースが，参照ペリフェラル仕様で定義されていて，通信の送受信機能，各カウンタ・タイマ機能に一つずつと，ソフトウェアによって直接生成されるもの(主に*FIQ*ハンドラをイネーブルして*IRQ*を生成するため)に一つ，という対応になっています．

◆カウンタ・タイマ

二つの16ビット・カウンタ・タイマが必要ですが，さらに追加することもできます．これらは*CTBase*から

の相対オフセットで指定されるレジスタで制御されます．カウンタはシステム・クロックで動作し，0，4，8ビットの選択可能なプリスケーラが付きます（したがって，入力周波数はシステム・クロック周波数割る1，16，256である）．

各カウンタ・タイマには制御レジスタがあり，それがカウンタのプリスケーリング，イネーブル，ディセーブルのいずれかの選択，動作モードがフリーランなのか周期的なのかを指定します．また，カウントが始まる値を指定するロード・レジスタもあります．「ロード」レジスタへの書き込みによってカウント値が初期化され，割り込みが生成するとゼロにデクリメントします．「クリア」レジスタへの書き込みは，割り込みをクリアします．フリーラン・モードではカウンタがゼロを超えてデクリメントを続けるのに対し，周期モードでは「ロード」レジスタの値で再ロードし，そこからデクリメントします．

現在のカウント値は，「値」レジスタからいつでも読むことができます．

◆リセットと一時停止コントローラ

リセットと一時停止コントローラには，*RPCBase*からの固定オフセットでアドレス指定されるレジスタがあります．読み出し可能レジスタは，識別およびパワーオンリセットが起こったかどうかなどのリセット状態情報を与えます．書き込み可能レジスタは，リセット状態をセットまたはクリア（ハードウェア・パワーオン・リセットでしかセットできないパワーオン・リセット状態ビットは除く），リセット・マップをクリア（たとえば，パワーオン後でARMリセット・ベクタのために必要な場合に，ROMをゼロ番地から通常メモリマップにスイッチするためなど），そしてシステムを一時停止モードにして割り込みが再びシステムを起こすまで最低限の電力しか使わないようにします．

◆システム設計

この基本コンポーネント・セットを組み込んだARMシステムはすべて，適切に設定されたOSカーネルをサポートします．後は，特定アプリケーション向け周辺装置やソフトウェアを加え，基本機能から積み上げていくだけです．

多くのアプリケーションがこれらのコンポーネントを要求するため，参照マイクロコントローラをシステム開発の出発点として使用するのにはほとんどオーバヘッドがかかりません．そのうえ，動作するシステムから出発するのは非常に有利です．

8.4　ハードウェア・システム・プロトタイピング・ツール

現在のシステム・オンチップ（SOC）設計者は，大変な仕事に立ち向かっています．チップ上のゲート数は急激に増加し続けていて，すでに何百万にも達しています．設計者は，市販されている最高のソフト設計ツールを使っても，そんなに複雑なシステムを完全にテスト済みのシステムとして製品化までの限られた時間内に生産することはできません．

この問題に対処する最初のステップは，すでに述べたように設計の大部分に既存の設計コンポーネントを使用することです．設計の再使用により，新しい設計作業をチップ上の総ゲート数のごく一部分に抑えることができます．AMBAなどのバスを使ったオンチップ相互接続へのシステマティックな方法に頼れば，設計作業はさらに削減することができます．しかし，それでも以下のように困難な問題があります．

(1) 選択した再使用可能ブロックが多様なソースからきているにもかかわらず，設計者はそれらが全部一緒に

正しく動作することをどう確認したらよいのでしょうか．

(2) 指定したシステムが，複雑なリアルタイム問題を含むことの多い性能要件を満たすことを，設計者はどう確認したらよいのでしょうか．

(3) ソフトウェア設計者は，チップを入手する前にどう仕事を進めたらよいのでしょうか．

ソフトウェア・ツールを使ったシステム・シミュレーションは，最終システムによるシミュレーションより性能が大幅に落ちることが多いため，ソフトウェア開発やフル・システム検証には役に立ちません．

これらの問題すべてに対応する非常に有効なソリューションは，ハードウェア・プロトタイピングです．つまり，必要なコンポーネントを全部組み合わせてハードウェア・システムを作るのです．これは最終システムの消費電力およびサイズに関する制約は満たしていませんが，たしかにシステム検証やソフトウェア開発のプラットホームにはなります．ARM「Integrator」は，そのようなシステムの一つです．また，VLSI Technology社の「ラピッド・シリコン・プロトタイピング(Rapid Silicon Prototyping)」も同様です．

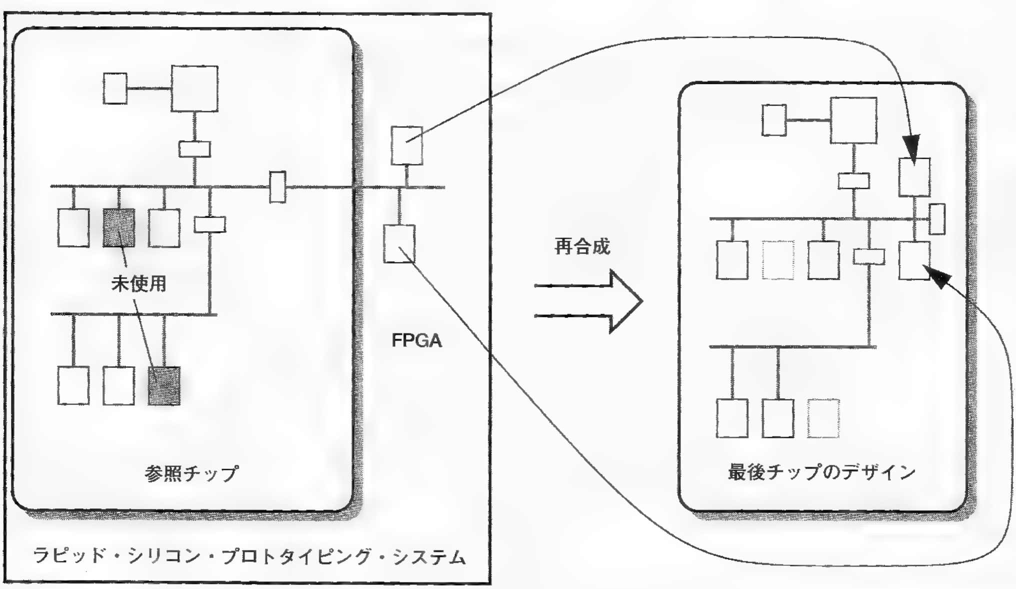
◆ラピッド・シリコン・プロトタイピング

VLSI Technologyは「ラピッド・シリコン・プロトタイピング」と呼ばれる開発システムを発表しました．このシステムの基本は，特別に開発された参照チップを使用することです．各チップには特定のオンチップ・コンポーネント群とオフチップ拡張のサポートが含まれていて，システム・オンチップ設計のプロトタイピングに使用できます．ターゲット・システムは，以下の二つの手順でモデリングします．

(1) 選択した参照チップを「構成解除」し，ターゲット・システムに不要なオンチップ・ブロックを非アクティブにします．

(2) 参照チップ上にはないが，ターゲット・システムに必要なブロックをオフチップ拡張として実装します．これは，必要な機能を備えた既存の集積回路のこともあれば，必要な機能に構成したFPGAのこともあります(通常はVHDLなどの高級言語による合成を使う)．

既存ブロックをAMBAなどの標準バスを使って相互接続すると，最終チップの生産にかかる技術的なリス



〔図8.13〕ラピッド・シリコン・プロトタイピング・パイプライン

クが最小限になります．参照チップ内の全ブロックは合成可能コンポーネントとして存在します．必要な機能を高級言語で記述したものとともに，FPGAの構成に使用した高級言語ソースを使用し，構成解除した機能を排除して再合成するとターゲット・チップができます．このプロセスは，図8.13に示したとおりです．

この方法にとっては明らかに，参照チップがCPUコアや信号処理システム（ターゲット・アプリケーションで必要な場合）など，適切なキー・コンポーネントを含んでいることが重要です．原則的には，1個の参照チップにあらゆるARMプロセッサ・コアを組み込むことが可能ですが（多様なキャッシュやMMU構成を含め），現実的にそのようなチップは，たとえプロトタイピング目的でも経済的ではありません．ですから，この方法が成功するためには，参照チップ上のコンポーネントを慎重に選択し，なるべく幅広いシステムに対応できるようにすることが大切です．そしてそれぞれのアプリケーション分野向けに，多様なCPUその他のキー・コアを含んだ各種参照チップを用意することです．

8.5 ARMulator

ARMulatorは，2.4節で説明したクロス開発ツール・キットの一部です．ARMプロセッサのソフトウェア・エミュレータで，ARMプロセッサ・チップを使用することなくデバッグとARMコードの評価をサポートします．

ARMulatorは，組み込み型システム設計において重要です．システム各部分のハイレベルなプロトタイピングをサポートし，ソフトウェア開発とアーキテクチャ選択の評価を支援します．ARMulatorは四つのコンポーネントで構成されています．

(1) Thumb命令セットを含め現在のどんなARMコアでもエミュレート可能な**プロセッサ・コア・モデル**．

(2) ターゲット・メモリ・システムの特徴をモデル化するための**メモリ・インターフェース**．迅速なプロトタイプ化をサポートするため，多くのモデルが多くありますが，インターフェースは必要なレベルの詳細を取り込めるよう完全にカスタマイズ可能です．

(3) カスタム・コプロセッサ・モデルをサポートする**コプロセッサ・インターフェース**．

(4) システム・コールごとにホストまたは，ARMモデルのエミュレーションで処理できるための**OSインターフェース**．

プロセッサ・コア・モデルは，リモート・デバッグ・インターフェースを組み込んでいるため，プロセッサおよびシステム状態はARMsd，つまりARMシンボリック・デバッガから見ることができます．このインターフェースを通して，プログラムのロード，動作，デバッグが可能です．

◆システム・モデリング

ARMulatorを使えば，キャッシュ，MMU，物理メモリ，周辺装置，OS，ソフトウェアを含むあるシステムについて，クロック・サイクル・ベースの精度のソフトウェア・モデルを構築することができます．これはもっともハイ・レベルのシステム・モデルであり，初期段階での設計の評価には最適といえます．

設計がある程度進めば，通常，ハードウェア開発はタイミング・ベースのCAD環境へ移動しますが，ソフトウェア開発はARMulatorベースのモデルを使い続けることができます（性能を向上するため，普通はサイクル・ベースから命令タイミング・ベースに移行する）．

ハードウェア設計が綿密になるにつれ，元のソフトウェア・モデルに組み込まれていたタイミング仮定が合わなくなることがあります．設計の進行にソフトウェア・モデルを合わせ，可能なかぎり正確なタイミング予測に基づいてソフトウェア開発をすることが大切です．

最近，ターゲット・システムを抽象化レベルの異なる複数コンピュータ・モデルを使って，複雑なシステム開発を行うことが多くなりました．この場合，抽象度の高いモデルから低いレベルのモデルを自動的に合成しない限り，モデル間の一貫性を維持するには，かなりの注意と努力が必要となります．

8.6 JTAGバウンダリ・スキャン・テスト・アーキテクチャ

特定アプリケーション向けシステム・チップを搭載した組み込みシステムの製品開発において困難なところは，VLSIコンポーネントのプロダクション・テストと，組み立てたプリント回路基板のプロダクション・テストといえます．

後者は，IEEE Std. 1149「スタンダード・テスト・アクセス・ポートおよびバウンダリ・スキャン・アーキテクチャ」によって対処されています．この規格は，ディジタル回路ピンの信号レベルにアクセス/制御する5ピン・シリアル・プロトコルについてのもので，チップ上での回路テストについても付記しています．この規格はJoint Test Action Group (JTAG) によって作られ，規格に書かれたアーキテクチャは「**JTAGバウンダリ・スキャン**」または「**IEEE1149**」として知られています．

JTAGバウンダリ・スキャン・テスト・インターフェースの一般的構造は図8.14のとおりです．コア・ロジックとピンの間の信号はシリアル・スキャン・パスがインターセプトします．シリアル・スキャン・パスは，通常の動作モードではコア・ロジックをピンにつなぎ，テスト・モードでは元の値を読み出して新しい値に変更することができます．

◆テスト信号

インターフェースとしては，テスト規格をサポートする各チップごとに備えなくてはいけない5本の専用信号をもっています．

(1) $\overline{TRST}$

テスト・リセット入力で，テスト・インターフェースを初期化します．

(2) *TCK*

テスト・クロックで，システム・クロックとは独立してテスト・インターフェースのタイミングを制御します．

(3) *TMS*

テスト・モード選択で，テスト・インターフェースのステートマシンの動作を制御します．

(4) *TDI*

テスト・データ入力線で，バウンダリ・スキャンや命令レジスタにデータを供給します．

(5) *TDO*

テスト・データ出力線で，バウンダリ・スキャン・チェインからサンプリングしたサンプル値を運び，シリアル・テスト回路内の次のチップへデータを伝播させます．

JTAGサポート付きチップ数個を組み込んだ基板上のテスト回路は，通常，$\overline{TRST}$，*TCK*，*TMS*を各チップに平列に接続し，一つのチップからの*TDO*を次のチップの*TDI*に接続することによって一つのループになるように構成されます．したがって，基板テスト・インターフェースは上記と同じ種類の5信号をもちます．

◆TAPコントローラ

テスト・インターフェースの動作は，テスト・アクセス・ポート (TAP) コントローラが制御します．これ

はステートマシンで，状態遷移は*TMS*が制御しています．状態遷移図は図8.15のとおりです．状態はすべて二つの出口をもつため，一つの信号，つまり*TMS*で遷移が制御できます．状態遷移図の二つの主要パスが，データ・レジスタ（DR）および命令レジスタ（IR）の動作を制御します．

◆データ・レジスタ

チップの動作は，テスト命令レジスタの内容によって決まります．テスト命令レジスタは，次のようにさまざまなデータ・レジスタを選択できます．

(1) デバイスIDレジスタ

チップにハード・ワイヤードされた識別番号を読み出します．

(2) バイパス・レジスタ

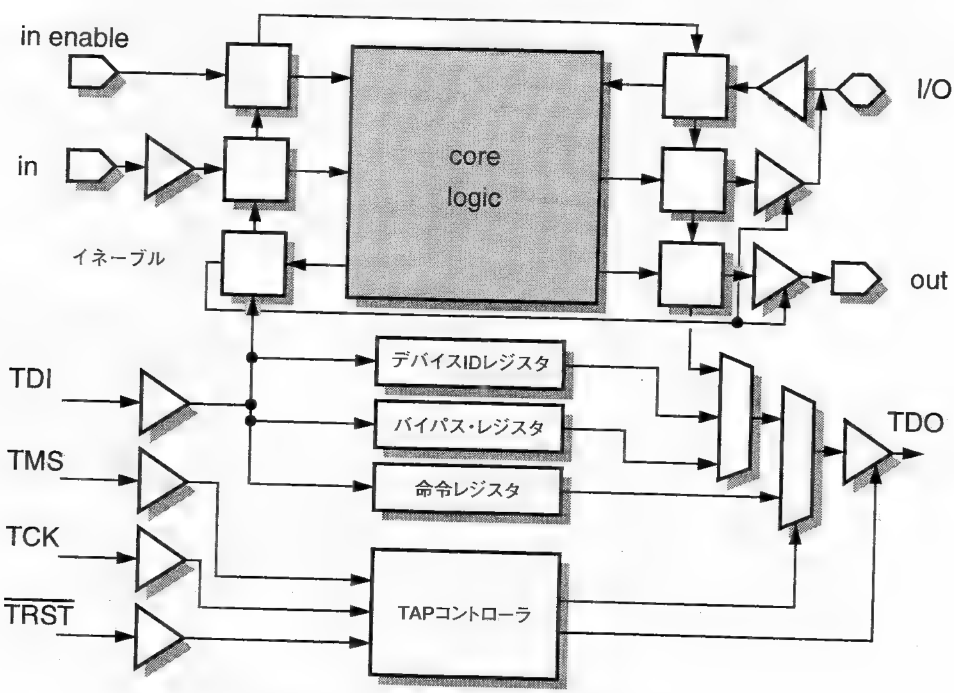
1クロックの遅延で*TDI*を*TDO*に接続するため，同じ基板上のテスト・ループにおいてテスタは他デバイスに高速アクセスできます．

(3) バウンダリ・スキャン・レジスタ

コア論理とピンの間の全信号をインターセプトし，図8.14でコア論理につながっている四角で示した個別のレジスタ・ビットを構成します．

(4) その他のレジスタ

他の機能をテストするため，必要に応じてチップ上に他のレジスタが採用されることもあります．



〔図8.14〕JTAGバウンダリ・スキャン構成図

◆命令

JTAGテスト・システムは次のように使用します．まず，次に実行するテストの種類，テストに使用するデー

タ・レジスタを指定する命令を「命令レジスタ」に入れ，そしてデータ・レジスタを使ってテストを実行します．

JTAGテスト用命令は，パブリック命令とプライベート命令があります．パブリック命令は，宣言されていて一般的テストに使用できます．規格では，規格準拠の全デバイスが対応すべきパブリック命令の最小限セットが指定されています．プライベート命令は，特別なオンチップ・テスト目的に使用するもので，規格では動作や用途について指定されていません．

◆パブリック命令

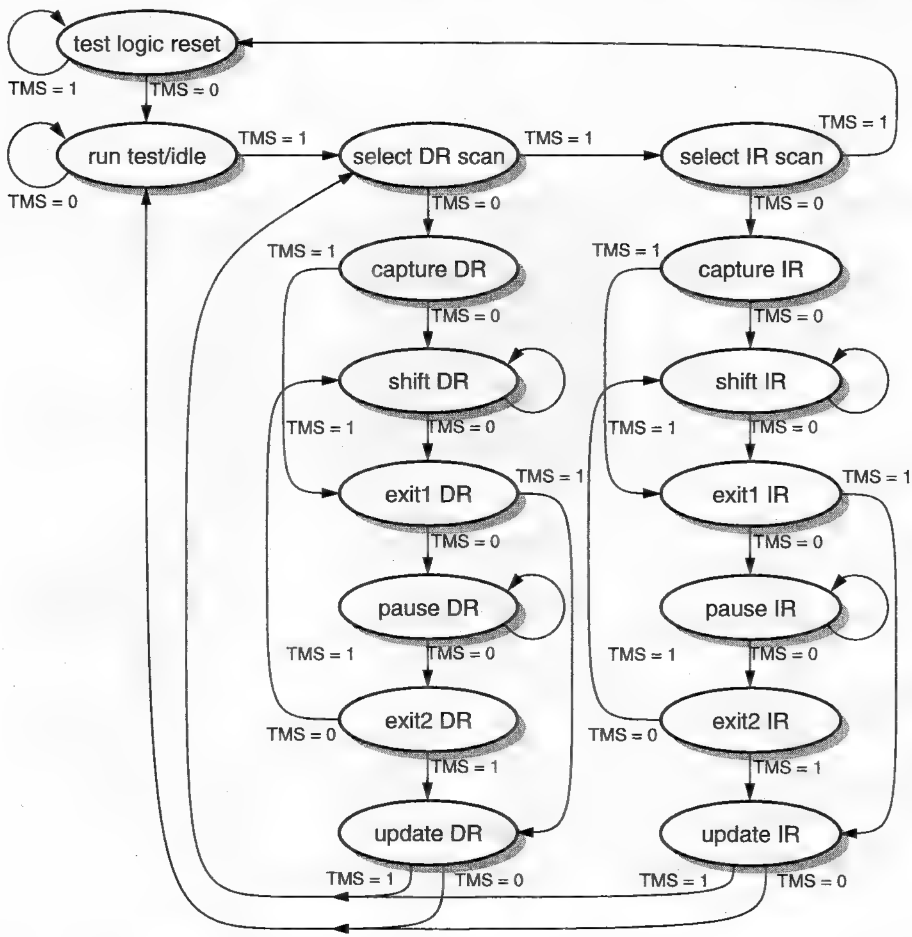
規格準拠デバイスが対応すべきパブリック命令の最小限セットは，次のとおりです．

(1) BYPASS

*TDI*と*TDO*を1クロック遅延で接続します．これは，同じテスト・ループ上の他デバイスをテストするための命令です．

(2) EXTEST

バウンダリ・スキャン・レジスタを*TDI*と*TDO*の間に接続し，レジスタがピン状態を取り込み，制御します．



〔図8.15〕 テスト・アクセス・ポート（TAP）コントローラ状態遷移図

図8.15の状態遷移図を見ると，*CaptureDR*状態でピン状態を取り込んで，*ShiftDR*状態で*TDO*ピンを通してレジスタ外へシフト・アウトしています．取り込んだデータをシフト・アウトすると，新しいデータを*TDI*ピン経由でシフト・インし，このデータを*UpdateDR*状態でバウンダリ・スキャン・レジスタ出力に（そして出力ピンへ）供給します．これは，基板レベル接続性をテストするための命令です．

(3) IDCODE

*TDI*と*TDO*の間に*ID*レジスタを接続します．*CaptureDR*状態でデバイスID（製造者，部品番号，部品バージョンを示すハード・ワイヤード識別番号）をレジスタにコピー，それを*ShiftDR*状態でシフト・アウトします．

その他の一般命令として，次の命令も含むことがあります．

(4) INTEST

バウンダリ・スキャン・レジスタを*TDI*と*TDO*の間に接続し，レジスタがコア論理入力と出力状態を捕らえ，制御します．入力が供給された値の補数を取ることに注目してください．それ以外は，EXTESTの動作とほぼ同じです．これは，コア論理をテストするための命令です．

◆PCBテスト

JTAGテスト回路の主な目的は，信号線の接続を可能にし，プリント回路基板（PCB）の動作不良（たとえば，はんだ接続不良）をテストすることです．これは，回路基板にスルーホールを必要としない表面実装パッケージの導入により，JTAGの手法が必要となりました．以前は「針山」のようなテスタでPCBの裏からICパッケージの全ピンを接続すれば，接続性をチェックすることが可能でした．しかし，表面実装パッケージはピン間が狭く，パターンも基板の部品面からしか届かないため，「針山」テスタが使えなくなったのです．

表面実装コンポーネントにJTAGテスト・インターフェースがあれば，これを使って（EXTEST命令を使用）出力を制御，チップの通常の機能とは関係なく入力を調べることが可能ですから，チップからチップへと基板レベルの接続性が楽にチェックできます．基板がJTAGインターフェースのないコンポーネントを含んでいる場合は，「針山」テスタが必要になりますが，これも使えるところはJTAGインターフェースと一緒に使い，コストもプロダクション・テスト装置を作る手間も最小限に抑えることができます．

◆VLSIテスト

複雑な集積回路になると，大がかりなプロダクション・テストを行って製品に組み込まれる前に欠陥デバイスを発見しなければなりません．プロダクションICテスタは非常に高価な装置で，各デバイスのテストにかかる時間は製造コストに影響する重要な要素です．JTAGテスト回路は，シリアル・アクセスで動作するので，テスト・ベクタをコア・ロジックに入力するのは速いとはいえません．そのうえ，JTAGポート経由でテスト・ベクタを適用してデバイスの通常動作速度でのデバイス性能をチェックすることもできません．

したがって，JTAGアーキテクチャがVLSIプロダクション・テストの全問題を解決する一般的な方法とはいえません．しかし，次のように多くの問題を解決します．

(1) JTAGポートはICのインサーキット機能テストに使うことができる（INTEST命令がサポートされている場合）．

(2) パラメータ・テストでICピンを制御できる（出力バッファのドライブ，漏れ，入力しきい値などをチェック可能）．これにはJTAG準拠デバイスすべてに必要なEXTEST命令だけを使う．

(3) 内部スキャン・パスにアクセスし，ピンからはアクセスが困難な内部ノードを観察，制御しやすくすることが可能．

(4) 追加ピンやシステム機能の妨害なしに，オンチップ・デバッグ機能にアクセスできる．これは，後でも簡

単に触れますが，8.7節で説明するARM EmbeddedICEデバッグ・アーキテクチャに採用されている.
(5) 後で説明するように，マクロセル・ベース設計の機能テストが可能.
上記の用途は，すべて，プリント回路基板のプロダクション・テストという主目的以外の用途です.

◆EmbeddedI-CE

8.7節で述べるARMデバッグ・アーキテクチャは，JTAGテスト・ポートの拡張に基づいています．EmbeddedICEモジュールは，特別なJTAG命令を用いて追加データ・レジスタとしてアクセスできるブレークポイント・レジスタおよびウォッチポイントを導入しています．さらに同様にアクセス可能なトレース・バッファも導入しています．ARMコア・マクロセル周辺のスキャン・パスは，システムの他の部分を妨害することなく命令をARMパイプラインに導入し，それらの命令はARMとシステム状態にアクセス，変更することに使用できます.

デバッグ・アーキテクチャは，複雑なシステム・チップ上のARMマクロセルをデバッグするうえで従来のICEシステム機能のほとんどをもっており，JTAGテスト・アクセス・ポートをデバッグ・ハードウェアの制御に使用しているため，チップ上にはピンを追加する必要がありません.

◆マクロセル・テスト

複雑なシステム・チップ設計において，多数の複雑な設計ずみマクロセルや，複数の特定アプリケーション向けカスタム論理を組み込む傾向が強まっています．ARMプロセッサ・コア自体もそのようなマクロセルです．他のマクロセルは，ARM社製もあれば，半導体パートナやサードパーティ・サプライヤのものということもあります．ですから，システム・チップの設計者がもつマクロセルの知識は限られており，各マクロセルのプロダクション・テスト・パターンについてはマクロセル・サプライヤに頼ることになります.

マクロセルは，システム・チップに埋め込まれているため，設計者は与えられたテスト・ベクタを各マクロセルに一つずつ適用していく方法を工夫しなければなりません．テスト・パターンは設計のカスタム論理部分用にも生成されなければなりませんが，設計者はその部分の論理も理解していると仮定されているのです.

マクロセルのエッジにテスト・パターンを入力するには，多くの方法があります.

(1) 各マクロセルからの信号を順に切り替えて，システム・チップの各ピンで出力するようなテスト・モードを備える.
(2) オンチップ・バスが，それに取り付けた各マクロセルへの直接テスト・アクセスをサポートする(8.2節を参照).
(3) 各マクロセルにバウンダリ・スキャン・パスがあり，それによりJTAGアーキテクチャの拡張を使ってテスト・パターンを適用する.

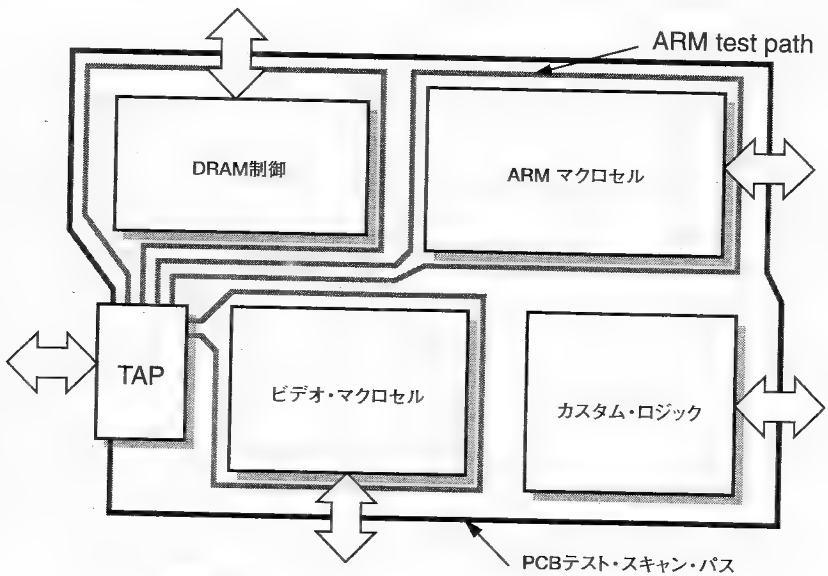
この最後の方法は，図8.16に示しています．チップにはパブリックなEXTEST操作に対応する周辺バウンダリ・スキャン・パスがあり，また，各マクロセル周辺にマクロセル内に設計された追加パスがあって供給された機能テストを適用します．このチップ向けに設計されたカスタム論理が独自のスキャン・パスをもっていてもよいし，図のように，インターフェース信号がすべて既存のスキャン・パスによってインターセプトされていることに頼ってもかまいません.

スキャン・パスを使ったマクロセル・テストは，機能テストとしてはたしかに有効ですが，JTAGバウンダリ・スキャン・パスを使ってチップのコア論理をテストするのと同様な弱点があります．シリアル・アクセスはピン経由のパラレル・アクセスよりかなり遅く，高速な性能テストが不可能なのです.

マクロセルに基づくシステム・チップに対して，もっとも有効となると思われるプロダクション・テスト方

法は，オンチップ・バスを使ってマクロセル周辺にパラレル・アクセスを可能にすることです（とくにこのルートでよくアクセスできるようにマクロセルが設計されている場合）．性能テストに重要でしかもオンチップ・バス経由ではアクセスに不便な周辺マクロセル信号に外部アクセスするには信号多重化を使い，必要に応じて他信号や内部状態にスキャン・チェイン経由でアクセスするにはJTAGポートを使用します．この方法は，8.2節で説明するARMの「Advanced Microcontroller Bus Architecture」（**AMBA**）によりサポートされています．AMBAテストについては本章の終わりで説明します．

JTAGシステムは，依然，基板レベルのテストでは重要です．また，コア・ロジックの入力回路テストやオンチップ・デバッグ装置へのアクセスにも使用されています．JTAGシステムは多くのARM設計に組み込まれ，テストおよびデバッグ方法の重要なコンポーネントとなっています．



〔図8.16〕マクロセル・テスト向けJTAG拡張例

8.7 ARMデバッグ・アーキテクチャ

コンピュータ・システムのデバッグは，複雑な作業です．デバッグには基本手法が二つあります．もっとも簡単なのは，ロジック・アナライザなどテスト装置を用いて外部からシステムを監視することで，もっとも強力なのはシングル・ステップ動作をするツールを用いてブレークポイントをセットして内部からシステムを見ることです．

◆デスクトップ・デバッグ

ターゲット・システムがデスクトップ・コンピュータ上で動作しているソフトウェアの場合，デバッガ自体も同じマシン上で動作するアプリケーション・ソフトウェアの一つであり，ユーザ・インターフェース・コンポーネントがすべてすぐに使えます．ブレークポイントをセットするには，オブジェクト・プログラム内の命令をデバッガ呼び出しに換え，実行がブレークポイントを過ぎれば再びもどせるように元の命令を覚えておきます．

コンパイラの中には，シンボル・テーブルなど詳しいデバッグ情報を生成するコンパイル・タイム・オプションがあります．デバッガがシンボル・テーブルを使うと，ユーザは変数をメモリ・アドレスでなくソース名でアドレス指定し，ソース・レベルの視点からプログラムをデバッグできます．この「ソース・レベル・デバッグ」は，オブジェクト・レベル・デバッグより強力で，マシン環境の知識もそれほど詳しく必要としません．

ソフトウェア・デバッガ共通の弱点は，「ウォッチポイント」機能の欠如です．ウォッチポイントとは，指定したメモリ・アドレスがデータ転送アドレスとしてアクセスされると実行を止めるものです．多くのプロセッサは特定アドレスのトラップをサポートしていないので（メモリ管理ページ・フォールトはこの目的には粗すぎるので除く），この機能は通常省略されていますが，これは残念なことです．Cプログラムで非常に多いエラーの原因は，プログラムの関係ない部分で間違ったポインタがデータを壊すことだからです．これをウォッチポイント機能なしで突き止めるのは非常に困難です．

◆組み込み型デバッグ

ターゲット・システムが組み込み型の場合，デバッグは非常に難しくなります．システムにユーザ・インタフェースがなかったり，デバッガはターゲットへの何らかの通信リンクを通して離れたホスト・コンピュータ上で動作しなければなりません．コードがROMにある場合は，その位置を書き換えることができませんから，命令を簡単にデバッガ呼び出しに替えることもできません．

ここでの標準的な解決策がICEです．ターゲット・システム内のプロセッサを取り外し，代わりにエミュレータへ接続します．エミュレータは同じプロセッサ・チップに基づいている場合もあれば，より多くのピンをもつ（内部状態がよく見える）バリエーションのこともありますが，バス動作を「トレース・バッファ」（全ピン上の信号を何サイクル分かストアする）にコピーするバッファ，ブレークポイントを通過する実行など特定の事象を見張る多くのハードウェア・リソースなどを組み込んでいます．トレース・バッファとハードウェア・リソースは，ホストのデスクトップ・コンピュータ上で動作するソフトウェアによって管理されます．

「トリガ」というイベントをかけると，トレース・バッファが停止し，ユーザはそのイベント周辺の動作を点検することができます．ホスト・ソフトウェアがトレース・バッファ・データを表示し，プロセッサやシステム状態を見せて変更を加えられるようにします．そして，デスクトップ・システム内のデバッガとできるだけ同じように見えるようにするのです．

◆プロセッサ・コアのデバッグ

ICEを使うには，システムにおいてプロセッサ・チップを識別し，取り外してICEと交換できなければなりません．プロセッサが複雑なシステム・チップ上にある多くのマクロセルの一つになってしまうと，これは明らかに不可能になります．

ARMulatorなどソフトウェア・モデルを使ったシミュレーションでは，物理的なシステムを組み立てる前に多くの設計バグを除去することができますが，エミュレーションで完全なソフトウェア・システムを動作することは普通不可能で，すべてのリアルタイム制約を正確にモデル化は困難です．ですから，何らかの方法で完全なハードウェアとソフトウェア・システムをデバッグすることが必要になってくるのです．どのようにすればいいのでしょうか？

ハードウェア・オーバヘッドを抑えた総合的な最善策については，まだ研究が行われている段階ですが，ここ2，3年，実践的な方法開発に大きな進歩がありました．本節では，ARM Limitedが開発した方法を紹介します．

◆ARMデバッグ・ハードウェア

ICEに対抗できるデバッグ装置にするには，ユーザがブレークポイントとウォッチポイントをセットし（RAMだけでなくROMで動作するコードについても），プロセッサおよびシステム状態を点検，変更し，問題箇所周辺のプロセッサ動作のトレースを見られるようにすることが必要です．また，すべてを快適なユーザ・インターフェースをもつデスクトップ・システムから操作できなければなりません．ARMシステムで使用されるトレース・メカニズムは，その他のデバッグ機能と分けて，8.8節で説明します．本節では，ブレークポイント，ウォッチポイント，ステート検査リソースに焦点を絞ります．

ターゲット・システムとホスト間の通信は，JTAGテスト・ポートの機能を拡張することで可能になります．JTAGテスト・ピンは基板テスト用として多くの設計に入っているため，このポートを通してデバッグ・ハードウェアにアクセスすれば専用ピンを追加する必要がなく，チップ上でもっとも貴重なリソースを節約できます．ブレークポイントおよびウォッチポイント・レジスタにアクセスしたり，プロセッサに命令してプロセッサとシステム状態にアクセスしたりするには，JTAGスキャン・チェインを使います．

ブレークポイントおよびウォッチポイント・レジスタは，ハードウェア・オーバヘッドが小さく，製品の一部として受け入れられます．ホストシステムは，シリアル・ポートとパラレル・ポート（またはその一方）でターゲットと通信しながら標準ARM開発ツールを実行します．ホストのシリアル線とターゲットJTAGポートの間には特別なプロトコル変換ハードウェアを設置します．

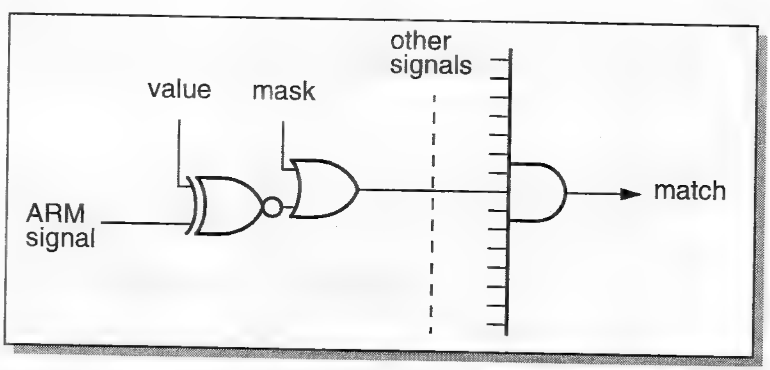
ブレークポイントおよびウォッチポイント事象に加え，システム・レベルの事象が起こった場合もプロセッサを止めるのが理想的です．このためデバッグ・アーキテクチャは外部入力をもっています．

これらの装置をもつオンチップ・セルのことを，Embedded-ICEモジュールと呼びます．

◆Embedded-ICE

Embedded-ICEモジュールは，二つのウォッチポイント・レジスタ，制御およびステータス・レジスタで構成されています．ウォッチポイント・レジスタは，プログラムされた値にマッチするアドレス，データ，制御信号を見つけるとARMコアを止めることができます．比較はマスク下で実行するため，どのウォッチポイント・レジスタも，ROMまたはRAM内の命令が実行されたときにプロセッサを止めるブレークポイント・レジスタとして動作するよう設定することもできます．

比較とマスクの論理は図8.17のとおりです．



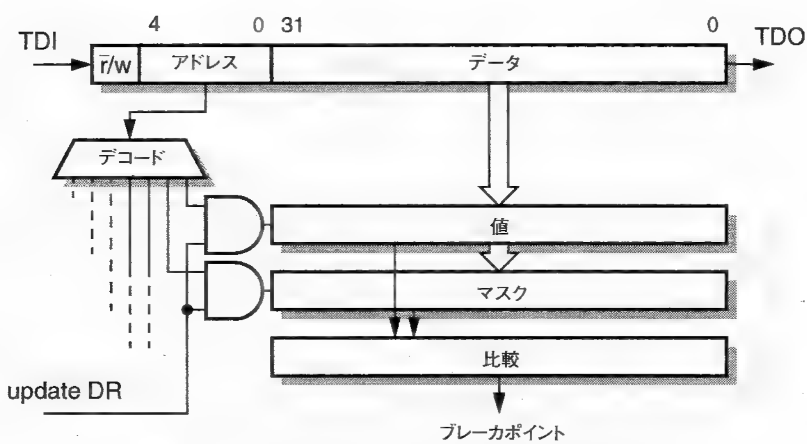
〔図8.17〕Embedded-ICE信号比較論理

◆チェイン

各ウォッチポイントは，ARMアドレス・バス，データ・バス，$\overline{trans}$，$\overline{opc}$，$\overline{mas[1:0]}$，$\overline{r}/w$制御信号の中から特定の値の組み合わせを探し，組み合わせが見つかるとプロセッサを止めます．もしくは，二つのウォッチ

〔表8.1〕Embedded-ICEレジスタ・マッピング

アドレス	幅	機能
00000	3	Debug control
00001	5	Debug status
00100	6	Debug comms control register
00101	32	Debug comms data register
01000	32	Watchpoint 0 address value
01001	32	Watchpoint 0 address mask
01010	32	Watchpoint 0 data value
01011	32	Watchpoint 0 data mask
01100	9	Watchpoint 0 control value
01101	8	Watchpoint 0 control mask
10000	32	Watchpoint 1 address value
10001	32	Watchpoint 1 address mask
10010	32	Watchpoint 1 data value
10011	32	Watchpoint 1 data mask
10100	9	Watchpoint 1 control value
10101	8	Watchpoint 1 control mask



〔図8.18〕Embedded-ICEレジスタの読み書き構造

ポイントをチェインしておき，最初のウォッチポイントがマッチした後で2番目がマッチしたときだけプロセッサを止めるようにすることもできます．

◆レジスタ

Embedded-ICEレジスタは，専用スキャン・チェインを用いてJTAGテスト・ポート経由でプログラムされています．スキャン・チェインは38ビット長で，32データ・ビット，5アドレス・ビット，レジスタが読まれるか書かれるかを制御する$r/\overline{w}$ビットをもっています．アドレス・ビットは，表8.1に示したマッピングにしたがって特定のレジスタを指定します．

JTAGスキャン・チェインの用途は図8.18のとおりです．読み書きが起こるのはTAPコントローラが「updateDR」状態に入るときです（図8.15参照）．

◆アクセス状態

Embedded-ICEモジュールは，プログラムを特定のポイントで止めることはできますが，直接プロセッサやシステム状態を点検，変更できるわけではありません．これは，JTAGポート経由でアクセスするもう一つのスキャン・パス経由で行います．

プロセッサ状態へアクセスするために使用するメカニズムは，まずプロセッサ停止，そして全レジスタのストア命令などをプロセッサの命令キューに設定することです．その後プロセッサにクロックが適用され，再びスキャン・チェイン経由で指示を与えプロセッサにデータ・ポートを通してレジスタ値を出力させます．各レジスタ値はスキャン・チェインによって収集され，シフト・アウトされます．

システム状態に関する情報収集はやや困難です．なぜなら，スキャン・パスが生成できる非常に遅い速度では読めない場所がシステム中にあるからです．しかし，この場合，プロセッサには適切な命令がプリロードされ，システム速度でその場所にアクセス可能です．これにより必要なシステム状態をプロセッサ・レジスタに転送し，同時に上記のようにJTAGポートを通じて外部デバッガへ渡すことができます．

◆デバッグ通信

ブレークポイントおよびウォッチポイント・レジスタに加え，Embedded-ICEモジュールには「デバッグ通信」ポートがあり，ターゲット・システムで動作するソフトウェアがホストと通信します．ターゲット・システムのソフトウェアは，通信ポートを6ビット制御レジスタと，コプロセッサ14へのMRCおよびMCR命令を使ってアクセス可能な32ビット・データ読み書きレジスタとして認識します．ホストはこれらのレジスタを表8.1のEmbedded-ICEレジスタ・マップで示すように認識します．

◆デバッグ

Embedded-ICEモジュールを含むARMベースのシステム・チップは，JTAGポートとプロトコル・コンバータでホスト・コンピュータと接続しています．この設定により，プログラマはネイティブまたはICEベースのデバッグで使いなれている通常のブレークポイント，ウォッチポイント，プロセッサおよびシステム状態へのアクセス（さらに，上記の通信ポート）を使うことができ，適切なホスト・ソフトウェアがあれば，低いハードウェア・オーバヘッドで完全なソース・レベル・デバッグ能力が得られるのです．

唯一欠けている機能は，リアルタイムでコードをトレースする機能だけです．これは次の節で述べる組み込みトレース・マクロセルが果たします．

8.8　組み込みトレース

リアルタイム・システムをデバッグする際，アプリケーション・ソフトウェアの動作をリアルタイムで観察する機能なしにデバッグするのはしばしば，困難です．Embedded-ICEマクロセルがもつブレークポイントやウォッチポイント機能は，この目的を果たすには不十分です．なぜなら，これらを使うとプロセッサは通常の実行シーケンスから離れるため，ソフトウェアのタイミング的な動作を壊してしまうからです．

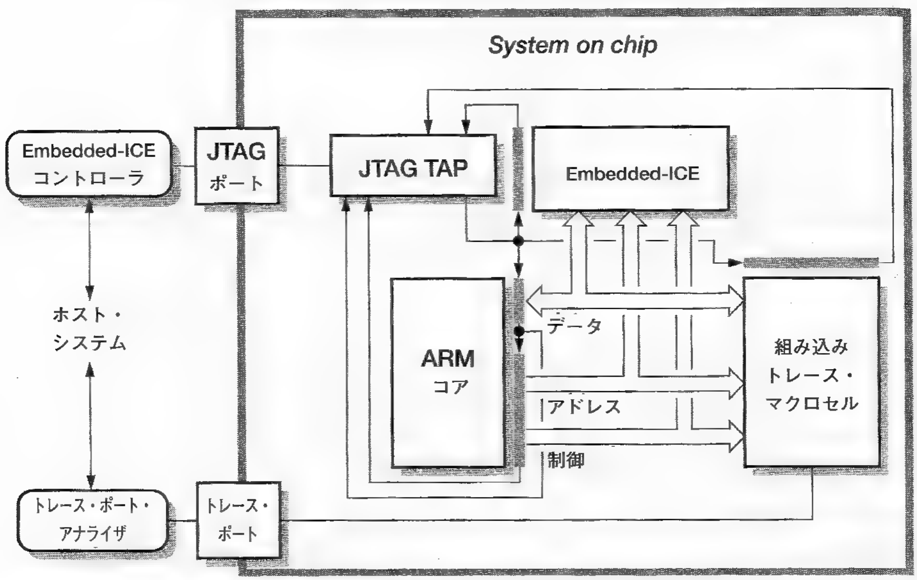
必要なのは，プログラムの実行中のアドレス，データ，制御バスそれぞれの動作のトレースを生成しながら，フルスピードで動作するプロセッサを観察する能力です．問題は，これには巨大なデータ帯域幅が必要になることです．100MHzで動作するARMプロセッサは，1Gバイト/秒のインターフェースを生成します．この情報をチップから得るには多数のピンが必要になるので，この機能を製品版のデバイスに組み込むのは非経済的です．したがって特別な開発デバイスが必要になりますが，新しいシステム・オンチップ・アプリケーションを開発するコストが高くなってしまいます．

◆トレース圧縮

ARM社が採用したのは，インテリジェントなトレース圧縮テクニックを使用してインターフェース帯域幅を低減するソリューションです．以下に例を挙げます．

(1) ほとんどのARMアドレスはシーケンシャルですから，全部のアドレスをチップ外に送る必要はありません．ほとんどのサイクルではシーケンシャル情報を送信し，フルアドレスは分岐があったときだけ送信するようにします．

(2) プロセッサ上で動作中のコードへアクセスできるオフチップの論理があれば，プロセッサがいつ分岐命令を実行するのか，分岐のターゲットがどこなのかがわかります．チップ外へ送信しなければならない情報



〔図8.19〕リアルタイム・デバッグ・システムの内部構成

は，分岐が起こったかどうかだけです．

(3) フルアドレス情報が必要なのは，サブルーチン・リターンやジャンプ・テーブル命令においてなど，分岐ターゲットがわからない場合だけです．この場合でも，変化する下位アドレスのビットのみ送信が必要です．

(4) アドレス情報は非常にバースト的です．必要なアドレス情報が4，8，16ビットのパケットで安定した速度で伝送されるように，FIFOバッファを使ってデータ転送速度を均一化することができます．

上記に述べたのと似たようなテクニックを多数用いることによって，ARM組み込みトレース・マクロセルはトレース情報を圧縮し，構成に応じて9，13，21ピンのいずれかを通じてオフチップと通信できる程度にします．これらのピンは，トレース出力が不要なときは別の目的に使用することができます．

◆リアルタイム・デバッグ

完全なリアルタイム・デバッグ・ソリューションは，図8.19に示したとおりです．Embedded-ICEユニットは，ブレークポイント，ウォッチポイント機能のほか，ホストとターゲット・ソフトウェア間の通信チャネルをサポートします．組み込みトレース・マクロセルは，プロセッサのインターフェース情報を圧縮し，トレース・ポートを通じてチップ外に送信します．両ユニットを制御するにはJTAGポートを使用します．また外部Embedded-ICEコントローラがホスト・システムをJTAGポートに接続し，外部トレース・ポート・アナライザがホスト・システムとトレース・ポート間のインターフェースとなります．ホストは，ネットワーク経由でトレース・ポート・アナライザとEmbedded-ICEコントローラの両方に接続することもあります．

ユーザは，ブレークポイントおよびウォッチポイントの設定と，さまざまなトレース機能を制御することができます．全部のアプリケーション・ソフトウェアをトレースすることもできれば，一部のルーチンだけでもかまいません．トリガ条件を指定することもでき，トレースはトリガの前でも後でも集められますし，トレースの途中にトリガが入ることもあります．データ・アクセスをトレースに含むかどうかも選択できますし，トレースがデータ・アクセスのアドレスだけを集めるようにも，データそのものだけを集めるようにも，その両方を集めるようにもできます．

◆組み込みトレースのオプション

上で述べたように，組み込みトレース・マクロセルは，いくつかの異なる構成で合成することができるため，ユニットの機能性とコスト（使用するゲート数とピン数から判断）のトレードオフを行うことができます．

(1) 最小のシステムには，パイプライン情報を発行する5ピンと，データを発行する4ピン（5ピンのJTAGインターフェースに加えて）が必要です．これで実行のトレースには十分ですが，非常に限られたデータ・トレースしかサポートしていませんし，トリガ機能とフィルタ機能も限られています．データ転送速度の均一化には9バイトのFIFOを使用し，この実装にかかるハードウェア・コストは約15Kゲートです．

(2) 最大のシステムには，パイプライン情報を発行する5ピンと，データを発行する16ピン（5ピンのJTAGインターフェースに加えて）があります．これで実行フローのトレースと，最悪時のデータ動作以外は全部トレースできます．データ転送速度の均一化には40バイトのFIFOを使用し，この実装にかかるハードウェア・コストは約50Kゲートです．

上記の両極端以外に，中間的な構成もいくつか可能です．どの場合も，トレースのトリガを制御する外部入力（つまりチップ上の別の論理からの入力）と，Embedded-ICEブレークポイント論理からのトリガに配慮しています．

◆トレース・オーバフロー

組み込みトレース・マクロセルを実装していても，トレースFIFOバッファがオーバフローする恐れのある状況がいくつかあります．オーバフローが起こった場合，組み込みトレース・マクロセルは，プロセッサをストールするようにもトレースを中止するようにも設定することができます．どちらにしても，FIFOの内容をはき出す間，リアルタイム・トレースは一時的に失われます．

基本的に何が起こったかというと，情報の帯域幅が，情報の帯域幅を縮めてトレース・ポートの帯域幅に収めるための圧縮アルゴリズムの能力を超えてしまったのです．これが起こった場合は，フィルタリングの設定を変更してトレースするデータの量を減らす必要があります．

◆Nトレース

リアルタイム・トレース技術は，ARM社，VLSI Technology社，Agilent Technologies社（もとはHewlett-Packardの一部）の協力によって開発されました．VLSI Technology社は「N-Trace」という製品名で組み込みトレース・マクロセルの合成可能バージョンを提供しています．

◆トレース・ポート・アナライザ

トレース・ポート・アナライザは従来からあるロジック・アナライザでもよいのですが，Agilent Technologies社はARM専用の安価なトレース・ポート・アナライザをすでに発売しており，その他のベンダも同様のシステムを発売する予定です．

◆トレース・ソフトウェア・ツール

組み込みトレース・マクロセルは，ARMソフトウェア開発ツールの拡張であるソフトウェアを使ってJTAGポート経由で設定します．トレース・データは，トレース・ポート・アナライザからダウンロードされ，ソース・コード情報を使って展開されます．そして，データ・アクセスを点在するアセンブリ・リスティングとして表示され，ソース・コードに戻るリンクも備えています．

Embedded-ICEと組み込みトレース・マクロセル機能によって，ARMシステム・オンチップ設計者は伝統的なインサーキット・エミュレーション（ICE）ツールが提供する全機能を利用できます．これらのテクノロジにより，アプリケーション・コードのリアルタイム動作が完全に見えるほか，ブレークポイント設定，プロセッサ・レジスタおよびメモリ位置の検査および変更が可能となります．また，高級言語ソース・コードへ戻るリンクも常に備えています．

8.9　信号処理サポート

ARMプロセッサをコントローラとして使用する多くのアプリケーションは，かなりのディジタル信号処理性能も必要とします．典型的なGSM携帯電話がよい例です．第1世代のARM搭載設計では，ARMコアと同じチップ上にDSPコアを組み込んでいるのが普通であり，システム設計者は，どのシステム機能をARMコア上に実装してどのシステム機能をDSPコア上に実装するほうがよいか慎重に選択をする必要があります．

DSPコアのプログラマ・モデルはARMコアのものと非常に異なります．DSPコアは何種類かのデータ・メモリを使用しており，最大スループットが得られるようプログラマがそれらの内部パイプラインを非常に注意深くスケジュールしなければならないことがよくあります．DSPコードを同時に動作しているARMコアと同

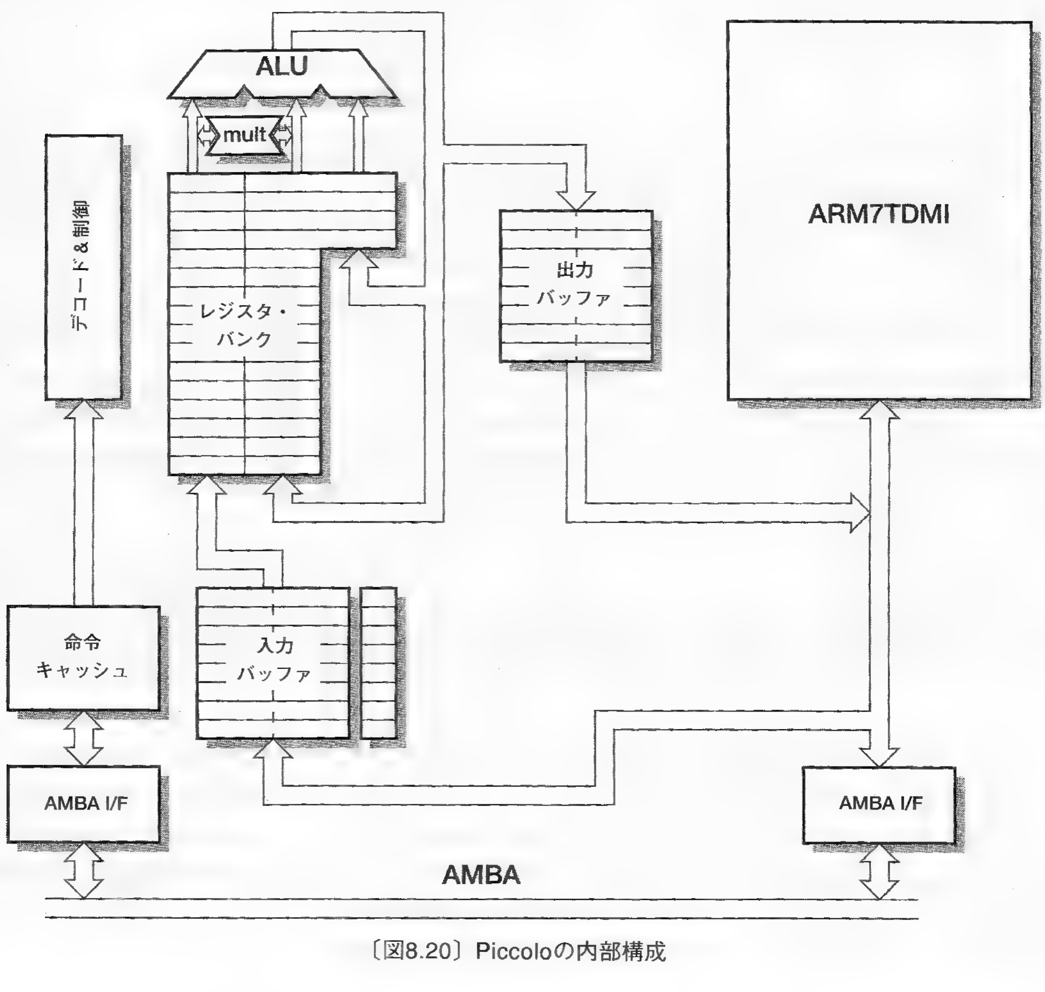
期させることは複雑な作業です．

ARM社は，ARMアーキテクチャに2種類の拡張を導入して，コントローラと信号処理機能の両方を要求するアプリケーションにおけるシステム設計作業を簡略化しようとしました．これがPiccoloコプロセッサとARMアーキテクチャversion5Tの信号処理命令セット拡張です．

◆Piccolo

Piccoloコプロセッサは，洗練された16ビット信号処理エンジンで，ARMコプロセッサ・インターフェースを使ってARMコアと共同で，オペランドと結果をメモリに(またはメモリから)転送するとともに，独自の命令セットも実行します．

Piccoloの構成は図8.20のとおりです．ARMコプロセッサ・インターフェース経由でオペランドをロードし，結果をストアしていますから，適切なアドレスはARMコアが生成する必要があります．入力および出力バッファにより，これらの転送では一つの命令で多数の16ビット値を転送することができ，ARMの32ビット・バス幅を活用するために値は二つ1組で転送します．各値は信号処理コードに呼び出されるまで入力バッファに



〔図8.20〕Piccoloの内部構成

ストアされ，バッファからは順序に関係なくそれらの値にアクセスできます.

Piccoloレジスタ・セットは，16，32，48ビット幅のオペランドを保持します．オーバフローの心配なく内部積などの結果を累積するには，四つの48ビット・レジスタを使用します．処理論理は，16×16積を計算し，1サイクルの間に，その結果をこれらの48ビット・アキュムレータ・レジスタのうちの一つに加算することができます．また，固定小数点演算に対する充実したサポートを備え，飽和演算もサポートしています.

レジスタ・ファイルに保持した値を使用する信号処理演算は，PiccoloがメモリからAMBAバス経由でローカル命令キャッシュにロードする別個の命令セットが指定されます.

Piccoloアーキテクチャの目的は，レジスタ，命令キャッシュ，入出力バッファの形で十分なローカルな記憶領域を確保し，一つのAMBAバスが高いスループットをサポートできるようにすることです．これは，二つの独立したデータ・メモリと別々の命令メモリを使用する従来の信号プロセッサ設計とは対照的です．このためPiccoloは，ARMコアと従来の信号処理コアを組み合わせたシステムより，ずっとわかりやすいプログラミング・モデルを提供するのです.

しかし，ARMコードとPiccoloコードを同期化するにはやはり多少の注意が必要です．集中的な信号処理アプリケーションでは，ARMコードはオペランドおよび結果転送に忙しいため，充実した制御機能を同時に実行することができません．ARMアーキテクチャversion5T命令セット拡張では，プログラミング・モデルがさらにわかりやすくなります.

◆version5TE信号処理命令

ARMアーキテクチャversion5TEで定義されている信号処理命令セット拡張は，最初，ARM9E-S合成可能コアに実装されましたが，Piccoloの設計とは大幅に異なるアプローチで問題に対処しています．使用しているのは，信号処理アプリケーションで使用するデータ型に対してもっと有効で本質的なサポートを提供するよう慎重に選択したネイティブARM命令セットへの追加だけです.

ARMプログラマ・モデルは，アーキテクチャversion5TEでは図8.21のように拡張しています．「Q」フラグがCPSRの27ビットに加わり，全SPSRもQフラグをもちます．Qは「スティッキ」なオーバフロー・フラグで，特定のversion5TE命令によってセットされ，適切なMSR命令によってリセットします(5.14節参照)．「スティッキ」という用語は，Qフラグが一度セットしたらMSR命令によって明示的にリセットされるまでセットされていることを表しています．そのため，一連の命令を実行してもQフラグは最後に一度点検(MRS命令を使う)してシーケンスの途中でオーバフローが起こったかどうか確認するだけですみます.



〔図8.21〕ARM version5TE PSRフォーマット

信号処理命令は，乗算，加算/減算という2グループに分けられます．加算/減算命令は飽和演算を使用します．つまり，結果がデータ型で表現される範囲からオーバフローした場合，表現可能な最も近い値が戻ります(そしてQフラグがセットされる)．これは，従来のプロセッサ演算(およびCで定義するモジュロ2^{32}演算データ・タイプ)とは対照的です．従来のプロセッサ演算では，最大値よりわずかに大きな結果は最小値に近い値を戻します．version5TEの場合，最大値よりわずかに大きな結果は最大値を戻すだけです．典型的な信号処理アルゴリズムにおいて，これはエラーを最小化し，最適な結果を出します.

◆version5TE乗算命令

乗算命令はすべて，プロセッサが16ビット・データ・タイプを処理する能力を向上し，32ビットARMレジスタが二つの16ビット値を保持すると仮定しています．したがって乗算命令は，レジスタの上位または下位半分に保持した値に対する効率的なアクセスを可能にします．乗算命令のバイナリ・エンコーディングは図8.22のとおりです．

31–28	27–23	22–21	20	19–16	15–12	11–8	7–4	3–0
cond	0 0 0 1 0	mul	0	Rd/RdHi	Rn/RdLo	Rs	1 y x 0	Rm

〔図8.22〕version5TEアーキテクチャの乗算命令のバイナリ・エンコーディング

この書式がサポートする命令は以下のとおりです．

```
SMLAxy{cond} Rd,Rm,Rs,Rn      ; mul = 00
```

この命令は，Rmの下位半分(x=0)または上位半分(x=1)とRsの下位半分(y=0)または上位半分(y=1)という二つの符号付き16ビット値の16×16積を計算します．32ビット積はRn内の32ビット値に加算され，結果はRdに置かれます．アセンブリ書式ではxとyの代わりに下位半ワードの場合は「B」，上位半ワードの場合は「T」を使用します．

```
SMLAWy{cond} Rd,Rm,Rs,Rn      ; mul = 01, x = 0
```

この命令は，Rm内の32ビット値とRsの下位半分(y=0)または上位半分(y=1)の32×16積を計算します．48ビット積の最上位32ビットはRnの32ビット値に加算され，結果はRdに置かれます．

```
SMULWy{cond} Rd,Rm,Rs         ; mul = 01, x = 1, Rn = 0
```

この命令は，Rm内の32ビット値とRsの下位半分(y=0)または上位半分(y=1)の32×16積を計算します．48ビット積の最上位32ビットはRdに置かれます．

```
SMLALxy{cond} RdLo,RdHi,Rm,Rs; mul = 10
```

この命令は，Rmの下位半分(x=0)または上位半分(x=1)とRsの下位半分(y=0)または上位半分(y=1)という二つの符号付き16ビット値の16×16積を計算します．32ビット積はRdHi:RdLo内の64ビット値に加算され，結果はRdHi:RdLoに置かれます．

```
SMULxy{cond} Rd,Rm,Rs         ; mul = 11, Rn = 0
```

この命令は，Rmの下位半分(x=0)または上位半分(x=1)とRsの下位半分(y=0)または上位半分(y=1)という二つの符号付き16ビット値の16×16積を計算します．32ビット積はRdに置かれます．

上記すべての命令において，CPSRフラグであるN，Z，C，Vは命令の影響を受けません．またPC(r15)は，どのオペランドや結果レジスタにも使用してはなりません．累積における加算(SMLAとSMLAW)でオーバフローが起こるとCPSRのQビットがセットされますが，加算は飽和演算ではなく従来のモジュロ2^{32}を使用しています．

◆version5TE加算/減算命令

アーキテクチャversion5TE拡張におけるもう一つの命令グループは，32ビット加算/減算命令であり，飽和演算を使用するものです．どの場合にも加算と減算を実行する前にオペランドの一つを2倍する命令が追加されており，これによって一部の信号処理アルゴリズムの効率が向上しています．加算/減算命令のバイナリ・エンコーディングは図8.23のとおりです．

31 28	27 23	22 21	20	19 16	15 12	11 8	7 4	3 0
cond	00010	op	0	Rn	Rd	0000	0101	Rm

〔図8.23〕version5TEアーキテクチャの加算/減算命令のバイナリ・エンコーディング

この書式がサポートする命令は以下のとおりです．

```
QADD{cond} Rd,Rm,Rn            ; op = 00
```

この命令は，Rm とRnの32ビット飽和加算を実行し，結果をRdに置きます．

```
QSUB{cond} Rd,Rm,Rn            ; op = 01
```

この命令は，Rmから Rnを引く32ビット飽和減算を実行し，結果をRdに置きます．

```
QDADD{cond} Rd,Rm,Rn           ; op = 10
```

この命令は，Rnを2倍(飽和付き)してから，その結果とRmの32ビット飽和加算を実行し，結果をRdに置きます．

```
QDSUB{cond} Rd,Rm,Rn           ; op = 11
```

この命令は，Rnを2倍(飽和付き)してから，Rmからその結果を引く32ビット飽和減算を実行し，結果をRdに置きます．

やはり，上記すべての命令において，CPSRフラグであるN，Z，C，Vは命令の影響を受けません．またPC(r15)は，どのオペランドや結果レジスタにも使用してはなりません．飽和加算や減算がオーバフローした場合，Rn(指定されていれば)を2倍することによってオーバフローが起こった場合はCPSRのQビットがセットされます．

◆version5TEサンプル・コード

上記の命令の用法を示すため，アーキテクチャversion5TE拡張をサポートしているARM9E-Sコアと，していないARM9TDMIにおいて，メモリに保持した16ビット符号付き数の2ベクタの内積(ドット積)を生成する場合の問題を考えてみましょう．信号処理アプリケーションにおいて内積の計算は非常に一般的な手続きです．エラーを最小化するためには飽和演算を使用する必要があります．中央ループのversion5TEコードは以下のとおりです．

```
loop    SMULBB  r3,r1,r2     ; 16x16 multiply
        SUBS    r4,r4,r2     ; decrement loop counter
        QDADD   r5,r5,r3     ; saturating x2 & accumulate
        SMULTT  r3,r1,r2     ; 16x16 multiply
        LDR     r1,[r6],#4   ; get next two multipliers
        QDADD   r5,r5,r3     ; saturating x2 & accumulate
        LDR     r2,[r7],#4   ; get next two multiplicands
        BNE     loop
```

このサンプル・コードはいくつかの重要なポイントを示しています．

(1) 命令は，パイプラインのストールを避けるように「スケジュール」されています．つまりARM9E-Sでは，ロードまたは16ビット乗算の結果は後続のサイクルで使用することができません．

(2) オペランドは16ビットハーフ・ワードですが，二つ1組で32ビット・ワードとしてロードします．これは，ハーフ・ワード・ロードを使うよりARMの32ビット・メモリ・インターフェースを効率的に使う方法です．また，version5TE乗算命令はレジスタから直接，個々の16ビット・オペランドにアクセスできます．

(3) 飽和「2倍して累積」命令は，累積の前に積をスケーリングするために使用されます．この命令が有用なのは，信号処理で使用する固定小数点演算では通常，オペランドが－1と＋1の範囲にあると仮定していますが，一部のアルゴリズムでは1より大きな係数が必要だからです．2倍演算によって－2と＋2という有効範囲を与えられ，ほとんどのアルゴリズムに十分となります．

◆性能比較

ARM9E-Sの単一サイクル32×16乗算器により，上記のループを10クロック・サイクルで完了することが可能となります．10のうち4サイクルはループ・オーバヘッドです（ループ・カウンタをデクリメントし，ループの最後で分岐して戻る）．各ループは，二つの積を計算しますから，各積には5サイクル必要です．ループ・アンローリング（1回のループ反復でさらに多くの積を計算するためのコード複写）を使えば，各積にかかるコストは3サイクルまで減ります．ARM9TDMIでは10サイクルで一つの積を出すのがせいいっぱいですから，3倍も遅いことになります．この違いを生むのは，ARM9TDMIの乗算器が遅いこと，16ビット・オペランドの処理が非効率的であることもありますが，ほとんどは飽和をテストして修正するのに余分な命令が必要になるためです．

8.10 例題と演習

<例題8.1>　JTAGおよびAMBAインターフェースを使って，ARMコアをテストする場合に必要なテストベクタ数を比べなさい．

ARMコアには，約100のインターフェース接続があります（32データ，32アドレス，制御，クロック，バス，モードなど）．JTAGインターフェースはシリアルです．テスタにより1ベクタがTCK上の1パルスを指定する場合，ARMコアにパラレルパターンを適用するには100ベクタかかります．AMBAテスト・インターフェースは5カ所でARM周辺装置にアクセス（本章の「テスト・インターフェース」を参照）するため，標準的テスタでは5ベクタが必要です．

したがって，JTAGインターフェースは20倍のベクタ数を必要とします（JTAGはPCBテスト向けであり，

VLSI製品テスト向けではないことを思い出してほしい）．実際は，JTAGに基づくEmbeddedICEとAMBAインターフェースは両方とも最適化を行い，ARMコアへ命令を入れる効率を向上しています．これは，テストのもっとも重要な目的ですが，この計算にそれを入れるのには非常に詳細な分析が必要になります．

演習8.1.1　複雑なマクロセル・ベースのシステム・チップのVLSI製品テストにおける問題分野についてまとめ，各解決策の長所を述べなさい．

演習8.1.2　VLSI製品テスト，プリント回路基板テスト，システム・デバッグの違いを記述し，JTAGテスト・ポートを使ってどのようにこれらに対処するか述べなさい．JTAG方式がもっとも効果的なところ，もっとも効果的でないところを指摘しなさい．

演習8.1.3　AMBA，ARM参照ペリフェラル仕様が対処しようとしている問題をあげなさい．それらはどのように関連していますか？

演習8.1.4　組み込み型システム・チップのシステム開発プランを書きなさい．そして，どの段階でARMulator，AMBA，ARM参照ペリフェラル仕様，Embedded-ICE，JTAGが(i)チップ上に設計されるか，(ii)開発プロセスに貢献するか，について(i)(ii)の片方もしくは両方を述べなさい．

第9章 ARMプロセッサ・コア

ARMコアは，ARM命令(Thumb)をメモリからフェッチして実行するシステム内のエンジンです．ARMコアは非常に小型で，通常は数mm^2のシリコン領域しか必要としません．現代のVLSIテクノロジにより，非常に多数の追加システム・コンポーネントを同じチップに組み込むことが可能となりました．これらには，キャッシュ・メモリ，メモリ管理ハードウェアなどのようにプロセッサ・コアに密接に関連しているものもあれば，信号処理ハードウェアのように無関係なシステム・コンポーネントもあります．さらに別のARMプロセッサ・コアを含んでいることすらあります．これら全コンポーネントの中でプロセッサ・コアは，もっとも高密度で複雑なコンポーネントとして際立っており，ソフトウェア開発やデバッグ・ツールに多大な要求をします．プロセッサ・コアの正しい選択は，新しいシステムの仕様においてもっとも重要な決断の一つです．

本章では，主要な現在のARMプロセッサ・コア製品について説明します．それらのコスト，複雑性，性能の程度はさまざまであり，そのうちもっとも効果的なソリューションを選択することができます．

多くのアプリケーションでは，プロセッサ・コアが緊密に結合したキャッシュおよびメモリ管理サブシステムにサポートされていることが必要です．これらのコンポーネントを組み合わせた標準的な構成は，第12章「ARM CPUコア」で多数紹介します．

さらにARM互換プロセッサ・コアについては，第14章「AMULET非同期ARMプロセッサ」で述べます．これらのプロセッサ・コアは研究用プロトタイプであり，まだ商業用製品ではありません．

9.1 ARM7TDMI

ARM7TDMIは，現行のローエンドARMコアであり，多くのディジタル携帯電話をはじめ，多様なアプリケーションで幅広く使用されています．

ARM7TDMIは，32ビット・アドレス空間プログラミング・モデルを最初に実装したARMコアであるARM6から進化しました．ARM6は，その回路テクニックにより5Vより小さい電源では安定した動作ができませんでした．ARM7ではこの欠点を改善し，さらに極めて短期間の間に64ビット乗算命令，オンチップ・デバッグ・サポート，Thumb命令セット，Embedded-ICEウォッチポイント・ハードウェアをすべて追加してARM7TDMIになりました．

名前の由来は以下のとおりです．

(1) ARM7. ARM6 32ビット整数コアの3ボルト互換改訂バージョン.
(2) Thumb 16ビット圧縮命令セット.
(3) オンチップ・デバッグ・サポート. デバッグ要求に応じてプロセッサの停止が可能.
(4) 強化型乗算器. 先行バージョンより高性能でフル64ビット結果を生成.
(5) Embedded-ICEハードウェア. オンチップ・ブレークポイントおよびウォッチポイント・サポートを提供.

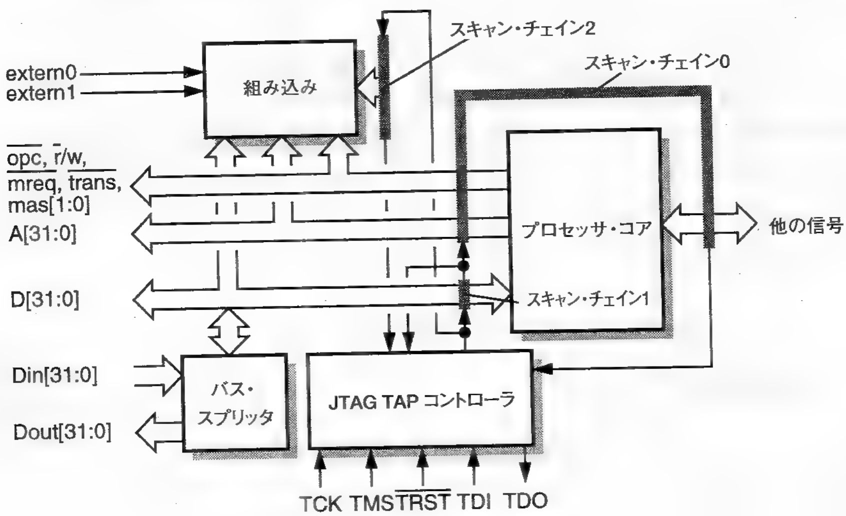
ARM7TDMIは, 多数の異なるCMOSプロセス・テクノロジによって製造されてきました. 100MHzを超すクロック速度をサポートするものもあれば, 0.9V (電源として単一セルのバッテリ使用が可能) で動作可能なものもあります. 本書執筆中の現在, 生産されている典型的なアプリケーションは, 0.35 μ mプロセスで3.3Vの電源を使用, クロック速度は最大66MHzですが, トレンドとしては, もちろんもっと小型のトランジスタ, 低電源電圧, 高クロック周波数に向かっています.

◆ARM7TDMIの構成

ARM7TDMIの構成は, 図9.1のとおりです. ARM7TDMIコアは, 3段パイプライン (4.1節参照) を使った基本ARM整数コアで, 多数の重要な機能や拡張を備えています.

(1) ARMアーキテクチャversion4Tを実装し, 64ビット結果乗算, ハーフ・ワードと符号付きロードおよびストア, Thumb命令セットをサポートする.
(2) Embedded-ICEモジュールを含み, 組み込み型システムのデバッグをサポートする (8.7節で詳しく説明).

JTAGテスト・アクセス・ポート経由でデバッグ・ハードウェアにアクセスするため, JTAG制御論理はプロセッサ・マクロセルの一部と見なされます (8.6節参照).



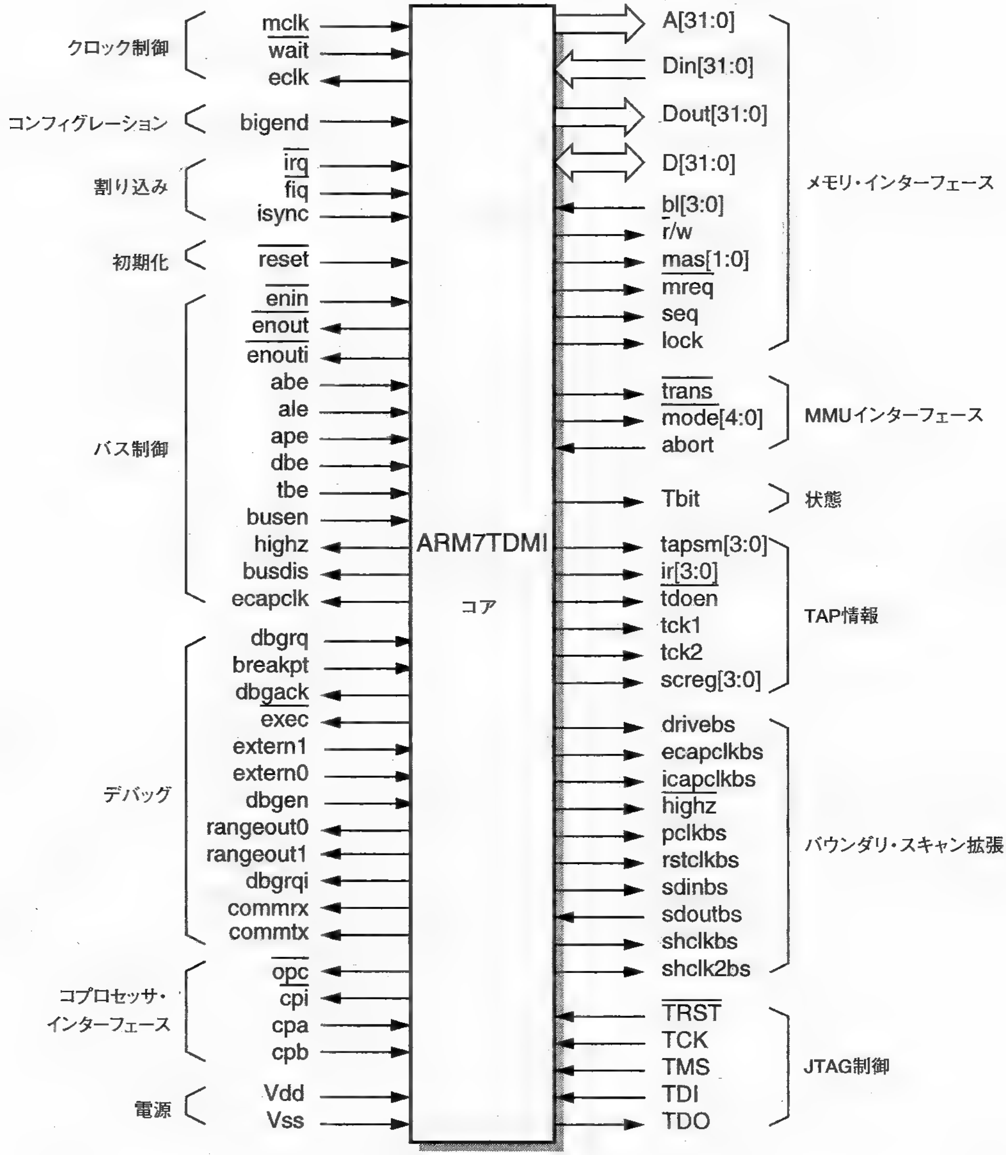
〔図9.1〕ARM7TDMI構成

◆ハードウェア・インターフェース

ARM7TDMIハードウェア・インターフェース信号は, 図9.2のとおりです. 驚くほどの数がある信号を見るだけでは誤解を招くおそれがあります. なぜなら, 信号数の多さは動作の複雑性につながり, 基本ARMインターフェース本来の単純性に反することになるからです. 数字的に見ると, インターフェース信号の大半は

基本32ビット・アドレス・バスおよびデータ・バスであり，下記のように，これらと少数の制御信号を単純なメモリ・インターフェースが使用しています．その他の信号は，オンチップ・デバッグ，JTAGバウンダリ・スキャン拡張など，もっと複雑な機能に使用します．

図9.2では，インターフェース信号を機能別に分類し，各グループの役割を下に記述しています．各信号とインターフェース・タイミングについての情報も適宜，記しています．



〔図9.2〕ARM7TDMIコアのインターフェース信号

◆クロック制御

プロセッサ内のすべての状態変化は*mclk*，つまりメモリ・クロックが制御します．このクロックを外部から操作してプロセッサに遅いアクセスを待たせることもできますが，たいていは，フリーラン・クロックをつけて$\overline{wait}$によってクロック・サイクルをとばすほうが簡単です．内部クロックは，事実上たんなる*mclk*と$\overline{wait}$の論理ANDですから，$\overline{wait}$は*mclk*が“L”の場合の間で変更できます．

eclkクロック出力は，コアが使用しているクロックを反映していますから，通常は$\overline{wait}$でゲートされたあとの*mclk*の動作を反映しますが，デバッグ・モードにおいてはデバッグ・クロックの動作も反映します．

◆メモリ・インターフェース

メモリ・インターフェースは，32ビット・アドレス(*A[31:0]*)，双方向データ・バス(*D[31:0]*)，別々のデータ・アウト(*Dout[31:0]*)およびデータ・イン(*Din[31:0]*)の各バスと10本の制御信号で構成されています．

(1) $\overline{mreq}$

プロセッサ・サイクルを示し，メモリ・アクセスを要求する．

(2) *seq*

メモリ・アドレスが前サイクルで使ったものとシーケンシャル(または，同じ)であることを示す．

(3) *lock*

プロセッサがバスを保持し，SWAP命令の読み出し・書き込みフェーズが不可分になるようにする．

(4) $\overline{r}/w$

プロセッサが読み出し，書き込みいずれかのサイクルを実行しているかを示す．

(5) *mas [1:0]*

メモリ・アクセス・サイズをエンコードし，アクセス単位がバイト，ハーフ・ワード，ワードのいずれかを示す．

(6) *bl [3:0]*

データ入力バスの4バイトのうち各バイトにおけるラッチの外部制御イネーブル．これらにより，32ビット幅未満のメモリのインターフェースが簡単になる．

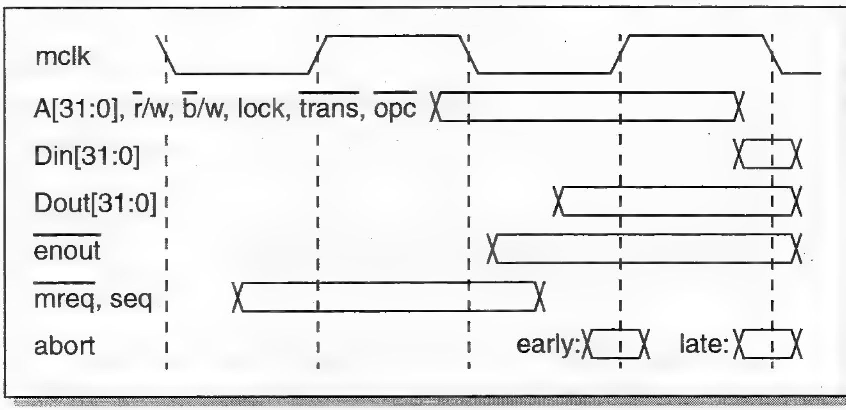
メモリ・サイクルのタイプを示す信号$\overline{mreq}$と*seq*は，メモリに制御ロジックが，メモリ・アクセスの処理方法を決めるためにできるだけ長く与えるように早期に発生します．これら二つの信号における，四つのとりうる値の組み合わせ解釈は表9.1のとおりです．

シーケンシャル・サイクルが非シーケンシャル・サイクルの後にくる場合は，アドレスは非シーケンシャル・サイクルのものに1ワード(4バイト)足したものになります．シーケンシャル・サイクルが内部またはコプロセッサ・レジスタ転送サイクルの後にくる場合は，アドレスは前サイクルと変わりません．一般的なメモリ構成では，高速シーケンシャル・アクセス用にメモリを準備するため，前アドレスに関する情報と一緒にインクリメント・ケースを使用します．アドレスが同じままの場合は，これを使って前サイクルのフルメモリ・アクセスを開始することができます(内部およびコプロセッサ・レジスタ転送サイクル両方ともメモリを使用しないため)．

クリティカルなインターフェース信号のタイミングは，図9.3のとおりです．信号の用途とメモリ・インターフェース論理の設計については，8.1節で例をあげて考えます．

〔表9.1〕ARM7TDMIのサイクル・タイプ

$\overline{mreq}$	*seq*	サイクル	用途
0	0	N	非シーケンシャル・メモリ・アクセス
0	1	S	シーケンシャル・メモリ・アクセス
1	0	I	内部サイクル — バス&メモリは非アクティブ
1	1	C	コプロセッサ・レジスタ転送 — メモリは非アクティブ



〔図9.3〕ARM7TDMIコアのメモリおよびMMUインターフェースのタイミング

◆MMUインターフェース

MMUへのインターフェース信号は，メモリ領域へのアクセスを制御するための情報を与えます．$\overline{trans}$（変換制御）信号は，プロセッサが

・ユーザ・モード（$\overline{trans}$=0）
・特権モード（$\overline{trans}$=1）

のどちらのモードかを示し，あるメモリ領域のアクセスをスーパバイザだけに制限したり，適切ならばユーザ・コードとスーパバイザ・コードで変換テーブルを使い分けること（めったにないが）ができます．動作モードについてのより詳細な情報が必要なときは，$\overline{mode[4:0]}$を用いるとCPSR（反転されている）の最下位5ビットを反映しています．ただし，このレベルのメモリ管理はほとんど用いられません．詳しいモード情報はデバッグの際にもっとも必要とされます．

アクセスが許可されない場合，*abort*入力信号を使います．*abort*のタイミングは，データとともにサイクルの終わりで有効になるようにします．これは図9.3に示されています．

メモリ・アクセスがアボートすると，アクセス中の$\overline{opc}$の値により，プロセッサはプリフェッチ・アボートまたはデータ・アボートを起こします．

MMUも，実行オンリのメモリ領域をサポートしなければならないときは$\overline{opc}$信号を用いることがあります．しかし，これはPC相対アクセスのためコード領域に保持しているリテラル・プールの使用を除外していることに注目してください．このため，実行オンリの保護はARMシステムでは広く用いられません（とくに，11.6

節で説明したARM MMUアーキテクチャではサポートされていない).

◆ステート

Tbit出力は，プロセッサが現在ARM命令を実行しているのかThumb命令を実行しているのかを，周辺に知らせます.

◆コンフィグレーション

bigendは，バイト順序をリトル・エンディアン，ビッグ・エンディアン間でスイッチします(エンディアンの説明については，「メモリ構成」を参照). この入力は，プロセッサの処理方法を設定するもので，動的に変更することは意図されていません. しかし，必要であればクロックのフェーズ2で変更できます.

◆割り込み

二つの割り込み入力は，プロセッサクロックと非同期でもかまいません. なぜなら，プロセッサの制御論理に入る前に同期化ラッチを通過してきているからです. 高速割り込み要求，$\overline{fiq}$は，通常割り込み要求，$\overline{irq}$よりも優先順位が高くなります.

*isync*入力により，環境がすでに*mclk*に同期した入力を与える場合には，割り込み同期を省略することができます. これにより割り込み遅延時間から同期遅延を除きます.

◆初期化

resetは，プロセッサを既知状態から起動します. アドレスは00000000_{16}から実行します.

◆バス制御

通常，ARM TDMIはできるだけ早く新しいアドレスを発し，MMUまたはメモリ・コントローラの処理時間を最長化するようにします. しかし，アドレス・バスが直接ROMやSRAMに接続している単純なシステムでは，古いアドレスをサイクルの最後まで保持しておかなければなりません. コアは*ape*で制御するトランスペアレント・ラッチを組み込んでおり，外部論理の要求があればアドレスをリタイミングすることが可能です.

ARM6コアは，書き込みサイクルを実行しているとき，$\overline{enout}$信号を発して知らせます. 外部データ・バスが双方向の場合，$\overline{enout}$は*dout[31:0]*をイネーブルするのに使われます. ときには，ほかのデバイスがバスを駆動できるよう書き込み動作を遅らせるほうがよい場合もあります. そのような場合，$\overline{enout}$がアクティブにならないようデータ・バス・イネーブル信号*dbe*を使うことができます. コアは，バスが使えるようになるまで停止しなければなりません($\overline{wait}$またはクロック延長を使います).

外部ロジックの要求があれば，*dbe*は外部からタイミングすることができます.

その他のバス制御信号である$\overline{enin}$，$\overline{enouti}$，*abe*，*ale*，*tbe*，*busen*，*highz*，*busdis*，*ecapclk*は，他のさまざまな機能を果たします. 詳しくは，ARM TDMIデータシートの対応箇所を参照してください.

◆デバッグ・サポート

ARM7TDMIは，8.7節で述べたARMデバッグ・アーキテクチャを実装しています. Embedded-ICEモジュールがブレークポイントおよびウォッチポイント・レジスタを含み，コードをデバッグ目的で停止させます. これらのレジスタはスキャン・チェイン2(図9.1参照)を使ってJTAGテスト・ポートで制御しています. ブレークポイントまたはウォッチポイントに行きつくとプロセッサが止まり，デバッグ状態に入ります. いったん

デバッグ状態になれば，スキャン・チェイン1を用いて命令パイプラインに命令を押し込むことによってプロセッサ・レジスタを点検できます．全レジスタの複数ストアによってデータ・バス上に値が現れるため，再びスキャン・チェイン1を用いてサンプルし，シフト・アウトすることができます．バンク・レジスタにアクセスするには，モード変更を起こすような命令が必要です（デバッグ状態では，通常と違ってユーザ・モードから特権モードにスイッチする際の障害がない）．

システム状態を点検するには，ARMをシステムの速度でメモリ位置にアクセスさせ，その後すぐにデバッグ状態にスイッチバックします．

◆デバッグ・インターフェース

デバッグ・インターフェースは，外部ハードウェアが（*dbgen*経由で）デバッグ・サポートをイネーブルして非同期デバッグ要求（*dbgrq*で）または命令同期要求（*breakpt*で）を出すことにより，統合Embedded-ICEマクロセルの機能を拡張するものです．外部ハードウェアは，コアがデバッグ・モードにあることを*dbgack*によって知ります．内部デバッグ要求信号は，*dbgrqi*でエクスポートします．

外部事象は，*extern0*と*extern1*経由でウォッチポイントのトリガを起こすことがあり，Embedded-ICEウォッチポイントのマッチは*rangeout0*と*rangeout1*で信号送信します．

通信送信バッファの空は*commtx*，受信バッファの空は*commrx*で信号出力されます．

プロセッサは，実行ステージにある現在の命令が実行されているのかどうかを$\overline{exec}$で表示します．命令は，条件コード・テストに合わなかった場合などは実行されません．

◆コプロセッサ・インターフェース

$\overline{cpi}$，*cpa*，*cpb*の各コプロセッサ・インターフェース信号は，4.5節で説明しました．他にコプロセッサに与えられる信号として$\overline{opc}$があり，メモリ・アクセスが命令をフェッチするかデータ項目をフェッチするかを示しています．これは，コプロセッサ・パイプライン・フォロワがARM命令実行をトレースするのに使います．コプロセッサを接続する必要がないとき，*cpa*と*cpb*は“H”につなぎます．この場合，全コプロセッサ命令が未定義命令トラップとなります．

◆電源

ARM6コアは，5Vまたは3V電源で動作するよう設計されていますが，これはコアに採用している回路設計スタイルと同様，プロセス技術に依存します．

◆JTAGインターフェース

JTAGインターフェースは，8.6節の標準規格に定めたとおりであり，専用ピン経由でオフチップのテスト・コントローラに接続しています．

◆TAP情報

これらの信号は，下記のバウンダリ・スキャン拡張とともに，JTAGシステムへさらにスキャン・チェインを追加するために使用します．

*tapsm[3:0]*は，TAPコントローラの状態を示します．*ir[3:0]*は，TAP命令レジスタの内容を伝えます．*screg[3:0]*は，TAPコントローラが現在選択しているスキャン・レジスタのアドレスです．tck1とtck2は，オーバラップしないクロック・ペアとして延長スキャン・チェインを制御しています．また，tdoenは，シリア

ル・データが*tdo*上に出力されるのかを表示します．

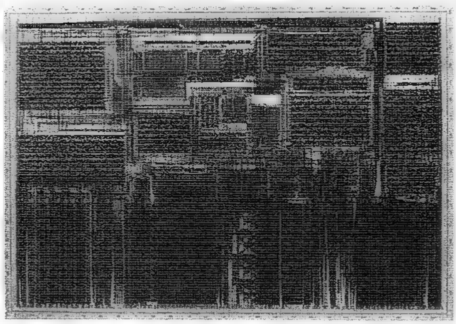
◆バウンダリ・スキャン拡張

ARM7TDMIには，Embedded-ICE機能をサポートするフルJTAG TAP（テスト・アクセス・ポート）コントローラが含まれています．このTAPコントローラを使えば，JTAGポートを通じてアクセスする他のオンチップ・スキャン機能をすべてサポートすることができます．したがって，システムに任意のスキャン・パスを追加できるよう*drivebs*，*ecapclkbs*，*icapclkbs*，$\overline{highz}$，*pclkbs*，*rstclkbs*，*sdinbs*，*sdoutbs*，*shclkbs*，*shclk2bs*の各インターフェース信号があります．これらの信号それぞれの機能について詳しくは，ARM7TDMIデータシートの該当箇所を参照してください．

◆ARM7TDMIコア

ARM7TDMIプロセッサ・コアの構成は，図9.4のとおりです．0.35μmコアで32ビットARMコードを実行するときの特徴は，表9.2にまとめてあります．

適切なプロセス・テクノロジを使えば，ARM7TDMIコアによって非常に高い電力効率を得ることができます．ある例では，0.9V電源で0.25μmプロセス・テクノロジを使って12,000MIPS/Wを実現しました．



〔図9.4〕ARM7TDMIプロセッサ・コア

〔表9.2〕ARM7TDMIの特徴

プロセス	0.6 μm	トランジスタ数	74,209	MIPS	36
メタル層	3	コア面積	5 mm^2	消費電力	80 mW
V_{dd}	3.3 V	クロック	0 to 40 MHz	MIPS/W	450

◆論理合成可能ARM7TDMI

標準のARM7TDMIプロセッサ・コアは「ハード」マクロセルです．つまり，然るべきプロセス技術にカスタマイズされた物理レイアウトとして出荷されています．ARM7TDMI-Sは，ARM7TDMIの合成可能バージョンで，高級言語モジュールとして出荷されターゲット・テクノロジの適切なセル・ライブラリを使って合成することができます．したがって，ハード・マクロセルより新しいプロセス技術への移植が簡単です．

合成プロセスは，プロセッサ・コア機能において多数のバリエーションをオプションとしてサポートしています．

(1) Embedded-ICEの省略．

(2) フル64ビット結果乗算器を，32ビットの結果を生成するARM乗算命令だけをサポートする小型で単純な乗算器に代える．

上記いずれかのオプションを選択することで，機能性を低減した小型の合成マクロセルを作ることができます．フル・バージョンは，ハード・マクロセルより面積が50%大きく，電力効率において50%劣ります．

◆ARM7TDMIアプリケーション

ARM7TDMIプロセッサ・コアは，通常は2-3Kバイトの単純なオンチップRAMを含んだシンプルなメモリ構成をもつシステムでよく利用されます．典型的な例は，携帯電話(通常，同じチップが高度なディジタル信号処理ハードウェアとそれ用のメモリを組み込んでいる)で，この分野ではARM7TDMIがその制御およびユーザ・インターフェース機能において事実上の標準プロセッサとなっています．

単純なメモリ・システムをもつARM7TDMIそのままより，もっと高性能が求められる場合は，システムが複雑にならざるをえません．まずはARM CPUマクロセルの形でARM7TDMIにキャッシュ・メモリを追加します．これによりオフチップ・メモリで動作するソフトウェアの性能が向上します．これでもまだアプリケーションに十分な性能が得られない場合は，もっと高性能レベルで動作する能力をもった複雑なARMコアを使用する必要があります．そのようなコアには，あとで説明するARM9TDMI，ARM10TDMIがあります．

9.2 ARM8

ARM8コアは，1993から1996年にARM Limitedで開発されました．ARM7の3段パイプラインよりも高性能のARMコアが求められていたからです．ARM8は今やARM9TDMI，ARM10TDMIに取って代わられましたが，その設計には興味深い点がいくつかあります．

4.2節で述べたように，プロセッサ・コアの性能を向上するには以下の方法があります．

(1) クロック速度の向上

各パイプライン・ステージの論理を単純化する必要があります．したがって，パイプラインの段数を増やします．

(2) CPI(命令ごとのクロック・サイクル)の低減

ARM7で二つ以上のパイプライン・スロットを占めていた命令を再実装して少ないスロットで足りるようにするか，命令間の依存性によるパイプラインのストールを減らすか，もしくはその両方が必要です．

◆CPIの低減

これも前に述べたことですが，ARM7よりCPIを低減することにおける根本的な問題は，ノイマン・ボトルネックにあります．つまり，単一の命令およびデータ・メモリをもつストアド・プログラム・コンピュータは，すべてメモリ幅によって性能を制限されるのです．ARM7コアは，(ほとんど)毎クロック・サイクルでメモリにアクセスし，命令をフェッチしたりデータ転送したりします．ARM7よりはるかに優れたCPIを得るには，各クロック・サイクルで二つ以上の値を引き渡さなければなりません．これには，単一メモリから毎サイクル32ビット以上を引き渡すか，または命令とデータ・アクセスで別々のメモリをもつことが必要です．

◆倍帯域幅メモリ

ARM8は，統合メモリ(キャッシュまたはオンチップRAMの形で)をもちますが，多くのメモリ・アクセス

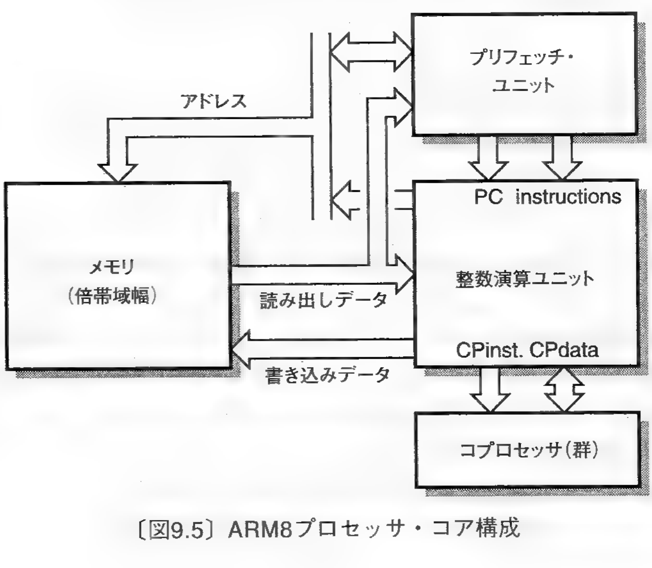
のシーケンシャルな性質を利用して，単一メモリからの「倍帯域幅（double bandwidth）」を可能にしています．これは，接続されているメモリが1クロック・サイクルに1ワード引き渡すことができ，次のシーケンシャル・ワードを半サイクル遅れで次のアクセスの開始と同時に引き渡すことができると仮定しています．一般的なメモリ構成では，少しハードウェア・コストを増やすだけで追加データを引き渡すことができます．増える帯域幅がシーケンシャル・アクセスのみに限定するのは使い道を狭めているように思えますが，命令フェッチはきわめてシーケンシャルで，ARMの複数ロード命令はシーケンシャル・アドレスを生成しますから（複数ストア命令も，ARM8では倍帯域幅メモリを使用していないが，やはり同様である），シーケンシャル・アクセスの発生頻度は一般的なARMコードではきわめて高くなります．

64ビット幅メモリは，必要とされる特徴をもっていますが，2番目のワードの到着を半クロック・サイクル遅らすと32ビット・バスを使うことができ，面積の節約になります．なぜなら32ビット・バスの配線は64ビット・バスの配線より少ない面積しか必要ないからです．

◆コア構成

ARM8プロセッサは，プリフェッチ・ユニットと整数データ・パスで構成されています．プリフェッチ・ユニットはメモリから命令をフェッチし，バッファする（倍帯域幅を利用するため）役割を果たします．そして，最大1クロック・サイクルごと1命令を，PC値と一緒に整数ユニットへ供給します．プリフェッチ・ユニットは分岐を予測する役目をもっており，分岐方向に基づく静的分岐予測（後ろ向き分岐は「分岐する」と予測され，前向き分岐は「分岐しない」と予測される）を使用しています．そして，整数ユニットが正確なストリームを計算，必要ならプリフェッチ・ユニットに訂正を送ります．

コアの全体的な構成は，図9.5のとおりです．倍帯域幅メモリは普通オンチップで，ARM810（12.2節に説明がある）など汎用デバイス上のキャッシュ・メモリ，または組み込み型アプリケーションのアドレス可能RAMの形をとっています．残りのメモリは従来の単一帯域幅デバイスを使っていますが，システムに何らかの高速メモリがなければ，単純なARMコアよりARM8コアが有利である点はほとんどありません．



〔図9.5〕ARM8プロセッサ・コア構成

◆パイプライン構成

プロセッサが使用する5段パイプラインでは，第1ステージをプリフェッチ・ユニットが占め，残りの4ステージを整数ユニットが占めます．

(1) 命令プリフェッチ
(2) 命令デコードとレジスタ読み込み
(3) 実行(シフトとALU)
(4) データ・メモリ・アクセス
(5) ライトバック結果

◆整数ユニット構成

ARM8整数ユニット構成は，図9.6のとおりです．命令ストリームと対応するPC値は，図の最上部に示したインターフェースを通してプリフェッチ・ユニットが供給します．システム制御コプロセッサは，図左の専用コプロセッサ命令，データの各バスを通して接続しています．データ・メモリへのインターフェースは図右にあり，アドレス・バス，書き込みデータ・バス，読み出しデータ・バスとなっています．

メモリ読み出しデータ・バスは，複数ロード命令で倍帯域幅転送をサポートしており，複数ロード命令で両レジスタ書き込みポートを使用，倍帯幅データ・ストリームをストアしていることに注目してください．複数ロード命令はワード量のみを転送するため，データ・ストリームの半分がバイト・アラインメントと符号拡張論理をバイパスしてもかまいません．

◆ARM8アプリケーション

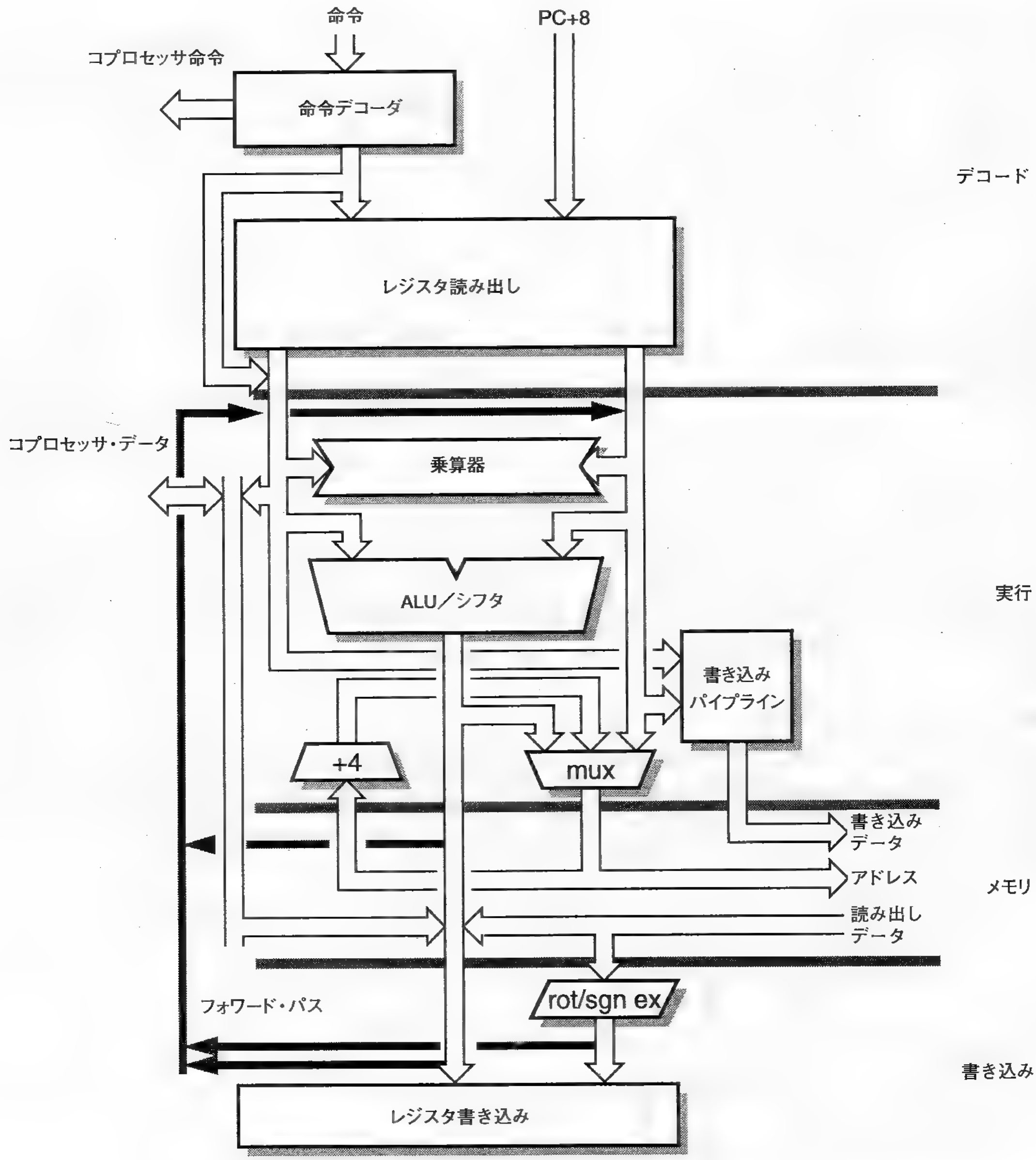
ARM8は，ARM Limitedのライセンシ各社がただちに製造できる汎用プロセッサ・コアとして設計されたため，特定のプロセス技術向けに最適化されているわけではありません．シリコン領域の増加が同程度の単純なARMコアに比べると，性能は大幅に向上(2～3倍)しています．潜在能力を最大に生かすためには，倍帯域幅オンチップ・メモリのサポートが必要です．

ARM8コアのアプリケーションの一つは，12.2節に説明するARM810など高性能CPUを構築することです．そこでは，倍帯域幅メモリはキャッシュの形をとっており，メモリ管理ユニットとシステム制御コプロセッサCP15もチップに組み込まれています．

◆ARM8シリコン

ARM8コアは，トランジスタ124,554個を使い，金属レイヤ3層をもつ0.5 μm CMOSプロセスで，最高速度72MHzで動作します．

コアのレイアウトは，図12.6のARM810のダイ写真でダイの左上の領域に見られるとおりです．



〔図9.6〕ARM8整数ユニット構成

9.3 ARM9TDMI

ARM9TDMIコアは，ARM7TDMIの機能性を大幅に高い性能レベルに引き上げたものです．ARM7TDMIと同様（ARM8とは違って），Thumb命令セットのサポートとオンチップ・デバッグをサポートするためのEmbedded-ICEモジュールを備えています．性能を向上させるため，5段パイプラインを採用して最大クロッ

密度を高めているほか，別々の命令メモリ・ポートとデータ・メモリ・ポートをもつことによってCPI（命令当たりのクロック数．1命令ごとにどれくらいのクロックがかかるかの測定単位）を向上させています．

◆性能向上

高性能を得るためにパイプラインの段数を3（ARM7TDMI）から5に増やす理由，メモリ・インターフェースを変更して別々の命令メモリ・ポートとデータ・メモリ・ポートを採用する理由については，4.2節を参照してください．

◆ARM9TDMIの構成

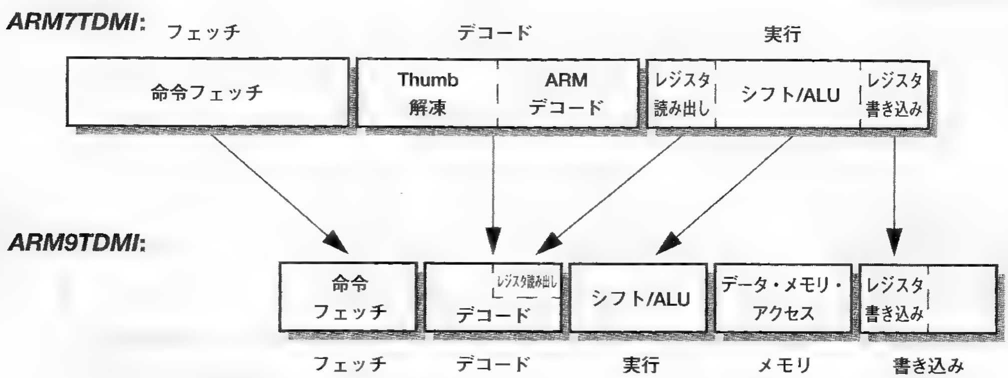
ARM9TDMIの5段パイプラインは，12.3節で説明するStrongARMのパイプラインに負うことが大きい（StrongARMは，スタンドアロン・コアとしての用途が限られているためコアとして本章に含まれていない）．

ARM9TDMIコアの構成は，図4.4のとおりです．ARM9TDMIとStrongARMコア（図12.8参照）の主要な違いは，StrongARMがレジスタ読み出しステージと平行して動作する専用の分岐加算器をもつのに対し，ARM9TDMIはメインALUを分岐ターゲットの計算に使用することです．これにより，ARM9TDMIでは，成立した分岐についてクロック・サイクル・ペナルティ一つが追加されます．しかし，コアは小型で単純になり，StrongARM設計に存在する非常にクリティカルなタイミング・パス（図12.9参照）を避けることができます．StrongARMは，このタイミング・パスが慎重に管理可能な特定のプロセス技術向けに設計されています．一方，ARM9TDMIは，そのようなクリティカル・パスが使用可能な最大クロック速度を簡単に下げてしまうような新しいプロセスへもすぐに移植できることが求められます．

◆パイプライン動作

ARM9TDMIの5段パイプラインの動作は，図9.7のとおりです．ここでは，ARM7TDMIの3段パイプラインと比較しています．この図は，プロセッサの主な処理機能を追加されたパイプライン・ステージに再配分して，同じプロセス・テクノロジでクロック周波数を約2倍にする仕組みを示しています．

この高いクロック速度を実現するのに必要なのは，実行機能（レジスタ読み出し，シフト，ALU，レジスタ書き込み）の再配分だけではありません．プロセッサは，ARM7TDMIの半分の時間で命令メモリにアクセスする必要がありますし，命令デコード論理も再構築して，レジスタ読み出しがデコードの大部分と同時に実行されるようにする必要があります．



〔図9.7〕ARM7TDMI&ARM9TDMI パイプラインの比較

◆Thumbのデコード

ARM7TDMIは，ARM7パイプラインのスラック（すき間）時間を使ってThumb命令をARM命令に「展開」することによってThumb命令セットを実装しています．ARM9TDMIパイプラインは非常に緊密なので，Thumb命令をまずARM命令に変換してからデコードするに足りるスラック時間がありません．その代わり，ARMおよびThumb命令の両方を直接デコードするハードウェアを備えています．

ARM9TDMIパイプラインに追加された「メモリ」ステージに直接相当するものは，ARM7TDMIにありません．この機能は，パイプライン・フローに割り込む追加の「実行」サイクルによって実行されています．この割り込みは，ARM7TDMIが命令アクセスとデータ・アクセスの両方に一つのメモリ・ポートを使用することによる避けられない結果です．データ・アクセスの間，命令フェッチは実行することができません．ARM9TDMIは，別々の命令メモリとデータ・メモリをもつことによって，このパイプライン割り込みを避けています．

◆コプロセッサ・サポート

ARM9TDMIはコプロセッサ・インターフェースを備えているため，浮動小数点，ディジタル信号処理，または特殊な目的のハードウェア化に対応するオンチップ・コプロセッサの実装が可能です（ARM9TDMIがサポートするクロック速度で，オフチップ・コプロセッサが動作する可能性はほとんどない）．

◆オンチップ・デバッグ

ARM9TDMIコアのEmbedded-ICE機能は，ARM7TDMIコアのものと同じシステムレベルのデバッグ機能を備えています（8.7節参照）．さらに，以下の特徴が加わりました．

(1) ハードウェア・シングルステップのサポート．

(2) ARM7TDMIがサポートするアドレス/データ/制御条件に加え，ブレークポイントを例外にセットすることが可能．

◆低電圧動作

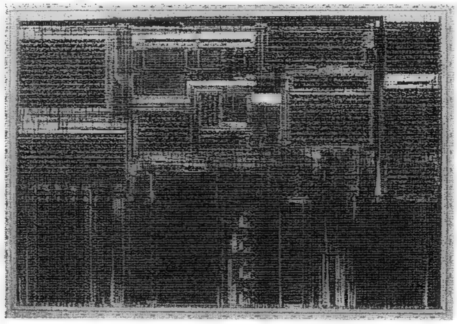
最初のARM9TDMIコアは，0.35 μm 3.3Vテクノロジに実装されました．この設計はこれまでに，最小1.2Vの電源で0.25 μmおよび0.18 μmプロセスに移植されています．

◆ARM9TDMIコア

32ビットARMコードを実行する際の0.25 μm ARM9TDMIコアの特徴は，表9.3にまとめてあります．コアの構成は図9.8のとおりです．

〔表9.3〕ARM9TDMIの特徴

プロセス	0.25 μm	トランジスタ数	111,000	MIPS	220
メタル層	3	コア面積	2.1 mm^2	消費電力	150 mW
V_{dd}	2.5 V	クロック	0–200 MHz	MIPS/W	1500



〔図9.8〕ARM9TDMIプロセッサ・コア

◆ARM9TDMIのアプリケーション

ARM9TDMIコアは，命令メモリ・ポートとデータ・メモリ・ポートを別々にもっています．これらのポートを一つの統合メモリに接続することは，原則的には可能ですが，そうなると，そもそも小型で安価なARM7TDMIよりARM9TDMIを選択する理由の多くがなくなってしまいます．同様に，ARM7TDMIの3段パイプラインに比べてARM9TDMIの5段パイプラインの高いクロック速度を採用する必要は必ずしもありませんが，そうしないのであればARM9TDMIを選択する理由がありません．したがって，ARM9TDMIコアを有意義に使用するアプリケーションはすべて，複雑な高速メモリ・サブシステムを扱わなければならないことになります．

このメモリ条件に対応するもっとも一般的な方法は，ARM9TDMIをベースとした多様な標準ARM CPUコアが示しているように，別々の命令キャッシュ・メモリとデータ・キャッシュ・メモリを組み込むことです．ARM920TとARM940T CPUコアについては，12.4節で述べます．これらのCPUコアにおけるキャッシュは，ARM9TDMIのメモリ帯域幅条件のほとんどを満たし，必要な外部帯域幅を減らして，一つのAMBAバス経由で接続した従来の統一メモリで満たせる程度にします．

性能クリティカルなコードを相対的に多く含むような組み込み型システムにとくに有効なもう一つの方法は，キャッシュの代わりに直接アドレスされるローカルな命令メモリとデータ・メモリを適度な量だけ使用することです．

◆ARM9E-S

ARM9E-Sは，ARM9TDMIコアの合成可能バージョンです．ARM9E-Sは，「ハード」コアに比べてARM命令セットの拡張バージョンを実装しています．ARM9TDMIがサポートするARMアーキテクチャversion4T命令に加え，ARM9E-Sは，8.9節で説明した信号処理命令セット拡張を含め完全なARMアーキテクチャversion 5TE（5.23節参照）をサポートしています．

ARM9E-Sは，同じプロセスを使った場合，ARM9TDMIより30%大きくなります．0.25μmCMOSプロセス上では2.7mm^{2}の面積を占めます．

9.4 ARM10TDMI

ARM10TDMIは，現在のハイエンドARMプロセッサ・コアであり，本書執筆時点ではまだ開発中です．同じプロセスにおけるARM9TDMIがARM7TDMIの約2倍の性能を発揮するのと同様，ARM10TDMIはARM9TDMIの2倍の性能をもちます．0.25μmCMOSテクノロジを使った場合，300MHzで400 dhrystone 2.1

MIPSを出すよう意図されています.

このレベルの性能を達成するには，ARM9TDMIを起点に二つのアプローチが組み合わされました（ここでも4.2節を参照）.

1. 最大クロック周波数の向上
2. CPI（命令当たりの平均クロック数）の低減

ARM9TDMIパイプラインがすでに極めて最適化されているのですから，どうすれば非常に複雑な構成，たとえばスーパースカラ実行（ARMコアの売り物である低電力と小さなコア面積を犠牲にすることになる）に頼らずに上記のような向上が得られるでしょうか？

◆クロック速度の向上

ARMコアがサポート可能な最大クロック速度は，全パイプライン・ステージで最も遅い論理パスによって決まります.

5段ARM9TDMIは，すでにうまくバランスがとれています（図9.7参照）. 5段のうち4段はいっぱいに詰まっています. パイプラインを拡張して論理をもっと多くのステージに分散することは可能ですが，そのような「スーパーパイプライン」構成の利点はたいてい，パイプライン依存性の増加によるCPIの悪化によって打ち消されてしまいます. そしてCPIの悪影響を最小限にしようとすれば，非常に複雑なメカニズムを導入せざるをえません.

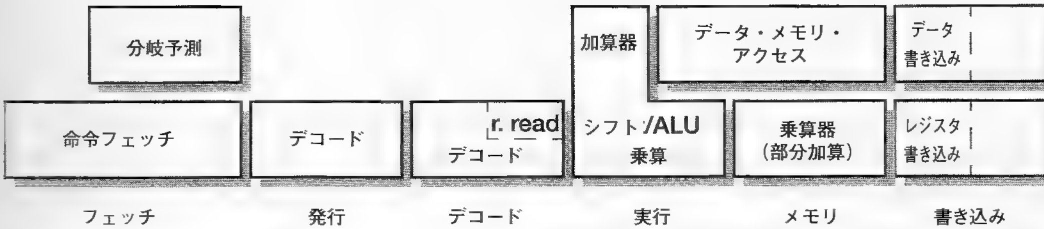
代わりにARM10TDMIは，ARM9TDMIと非常に似たパイプラインを維持しながら各ステージを一定の方法で最適化することによって高いクロック速度をサポートしています（図9.9）.

(1) 次サイクルのアドレスを早く与えることによって，フェッチ・ステージとメモリ・ステージを1から1.5クロック・サイクルに効果的に延長します. メモリ・ステージでこれを実現するために，メモリ・アドレスは主ALUより速く結果を生成できる別個の加算器（これは，ALUの機能のサブセットだけを実装しているため）で計算します.

(2) 実行ステージは，向上した回路テクニックと再構成によってクリティカル・パスを低減しています. たとえば乗算器は，部分和を積の項に分解するためにALUに値を入力しません. 代わりに，メモリ・ステージに独自の加算器をもちます(乗算はメモリにアクセスすることがないため，このステージは空きになる).

(3) 命令デコード・ステージは，プロセッサ論理の中で十分にストリームライン化してクロック速度を上げることのできなかった唯一の部分です. ですから，ここに新しく「発行」パイプライン・ステージが挿入されました.

この結果，5段ARM9TDMIより動作の速い6段パイプラインが生まれましたが，それをサポートするメモリはARM9TDMIのメモリより少し速い必要があります. 非常に高速なメモリは，電力を多量に消費する傾向がありますから，これは重要なことです. 挿入した新しいパイプライン・ステージにより，命令デコードに時間の余裕が生まれますが，予測していない分岐が実行されるとパイプライン依存性による弊害があることだけが問題です. この追加ステージはレジスタ読み出しが実行される前にあるので，新しいオペランド依存性は起こらず，新しいフォワーディング・パスは不要です. 分岐予測メカニズムを含んだこのパイプラインでは，高いクロック速度をサポートしつつ，ARM9TDMIパイプラインに非常に近いCPIが得られます.



〔図9.9〕ARM10TDMIパイプライン

◆CPIの低減

上記に説明したパイプライン拡張は，CPIを悪化させることなく50%も高いクロック速度をサポートします．これは上々な出だしですが，100%の性能向上を生み出すには至りません．このため，クロック速度の向上に加えてCPIの向上が必要となります．

CPIを向上する計画は常に，メモリ帯域幅を考慮することから始まります．ARM7TDMIは（ほとんど）毎クロック・サイクルで単一の32ビット・メモリを使用しますから，ARM9TDMIではハーバード・メモリ構成にしてさらに帯域幅を広げました．ARM9TDMIはその命令メモリを（ほとんど）毎クロック・サイクルで使用します．データ・メモリは約50%の負荷しかありませんが，これをCPI向上に利用するのは困難です．どうにかして命令帯域幅を広げる必要があります．

ARM10TDMIで採用した方法は，64ビット・メモリの使用です．これは命令帯域幅のボトルネックを効果的に解決し，CPIを向上する特長を数多くプロセッサ構成に加えることが可能になります．

1) 分岐予測．ARM10TDMI分岐予測論理は，上記で述べたようにパイプラインの効率性を単に維持するのに必要なレベルを超えています．なぜなら命令がクロック・サイクル当たり2という速度でフェッチされるうえ，分岐予測ユニット（フェッチ・パイプライン・ステージにある）は分岐を発行される前に認識できることが多く，それらを効果的に命令ストリームから除去します．そのため，分岐のサイクル・コストがゼロにまで低減できるのです．

ARM10TDMIは静的な分岐予測メカニズムを採用しています．後ろ向きに分岐する条件分岐は成立したと予測され，前向きに分岐するものは不成立と予測されます．

2) ノンブロッキング・ロードおよびストア実行．低速メモリを参照したり複数のレジスタを転送したりする理由によって1メモリ・サイクルで完了できないロードまたはストア命令は，オペランド依存性が発生しない限り実行パイプラインをストールしません．

3) 64ビット・データ・メモリにより，複数ロード命令および複数ストア命令が各クロック・サイクルで2レジスタを転送できます．

ノンブロッキング・ロードおよびストア論理には，独立したレジスタ・ファイル読み出しポートと書き込みポートが必要です．また64ビットの複数ロード命令および複数ストア命令には，それが二つずつ必要です．そのためARM10TDMIレジスタ・バンクには，四つの読み出しポートと三つの書き込みポートがあります．

これらすべての特長によって，ARM10TDMIはdhrystone2.1 MIPS/MHzの値1.25を達成しています．この値は，ARM7TDMIで0.9，ARM9TDMIで1.1となっています．dhrystoneベンチマークを動作すると，これらの数値は各CPI性能を直接反映します．その他のプログラムは多少異なるCPI結果を出すかもしれません．また，ARM10TDMIは64ビット・データ・バスのおかげで，オペレーティング・システムの起動など複雑な作

業においてARM9TDMIより大幅によい実効CPIを発揮することができます．

◆ARM10TDMIのアプリケーション

「ARM9TDMIのアプリケーション」で，コアの潜在的な性能をすべて発揮するには，少なくともいくらかのローカルな高速メモリが必要であると述べました．ARM10TDMIでも同じことです．別々になったローカルの64ビット命令メモリおよびデータ・メモリがなければ，コアはその性能を完全に発揮しません．速度も，もっと小型で安価なARMコアより速くなることはないでしょう．

やはり，この問題を解決する通常の方法（唯一の方法ではないが）は，12.6節で説明するARM1020Eが示すように，ローカルなキャッシュ・メモリを備えることです．ARM10TDMIコアの性能は，高速64ビット・ローカル・メモリの有無に決定的に依存しますから，その性能特性についてはARM1020Eについて述べる中で紹介します．

9.5　考察

ARM7TDMIまでの初期のARMプロセッサ・コアはすべて，単純な3段フェッチ-デコード-実行パイプラインに基づいていました．1980年代前半にAcorn Computersで開発された初代ARM1から，現在の携帯電話のほとんどに搭載されているARM7TDMIコアまで，動作の基本原則はほとんど変化していません．ARM最初の10年間に行われた開発事業は，設計の以下の点に焦点を当てていました．

①クリティカル・パスの最適化とプロセス縮小による性能向上

②スタティックCMOS論理による低消費電力アプリケーション，電源電圧低減，コード圧縮（Thumb命令セット）

③オンチップ・デバッグ機能，オンチップ・バス，ソフトウェア・ツールの追加によるシステム開発のサポート

ARM7TDMIは，この開発過程の絶頂を示しています．その商業的な成功は，スーパースカラ，スーパーパイプラインなどを備えた限りなく複雑な高性能（かつ電力消費の大きい）マイクロプロセッサをもつPCが支配的な世界で，基本的な非常に単純な3段パイプラインが通用することを実証しています，

ARM開発の次の10年間には，高性能レベルを追求してARM構成の慎重な多様化が進みました．

(1) 5段パイプラインへの第一歩では，コア内のフォワーディング論理と，2倍の帯域幅をもつメモリ（ARM8）や，別々の命令メモリとデータ・メモリ（ARM9TDMIおよびStrongARM）によって，性能が2倍になりました（その他の条件はすべて同じとする）．

(2) 次に性能を2倍にしたのはARM10TDMIでしたが，これはなかなか困難でした．6段パイプラインは前に使っていた5段パイプラインと極めて似ていますが，メモリ・アクセスに当てるタイム・スロットを延長することにより，多量の電力を消費することなくメモリが高いクロック速度をサポートできるようになっています．プロセッサ・コアには数々の分割化も採用されました．たとえばプリフェッチ・ユニットでは，分岐を予測して命令ストリームから除去するため，データ・メモリ・インターフェースでは，データ・アクセスの完了に時間がかかる際に（キャッシュ・ミスなどで）プロセッサが実行を継続するためです．

性能向上は，クロック速度の向上とCPI（命令当たりの平均クロック数）の低減によって実現しました．クロック速度を向上すると通常，深いパイプラインが必要となりCPIを悪化させますから，CPIの低下を回復して

さらに向上させる対策が必要です．

現在までの全ARMプロセッサは，1クロック・サイクル当たり最大1命令を常にプログラム順序で発行する構成をベースにしています．ARM10TDMIとAMULET3プロセッサ（14.5節で述べる）では，アウト・オブ・オーダの完了を処理することによって，遅いデータ・アクセスの間も命令が流れ続けるようにしています．両プロセッサとも分岐予測論理を含み，分岐命令の際にパイプラインを詰めなおすコストを節約しています．AMULET3は，予測される分岐命令のフェッチを抑制しますが，やはり実行します．ARM10TDMIは，分岐命令をフェッチしますが，実行を抑制します．しかし，今日のハイエンドPCやワークステーション・プロセッサの水準からいえば，これらはまだ非常に単純なマシンです．この単純さこそ，システム・オンチップのアプリケーションに直接的に有利なのです．単純なプロセッサは，複雑なプロセッサより少ないトランジスタ数ですむため，ダイ面積も小さくて済み，消費電力も少なくなるからです．

9.6 例題と演習

＜例題9.1＞ スタティックRAMやROMデバイスにインターフェースするために，どのようにARM7TDMIアドレス・バスをリタイミングしますか？

通常，ARM7TDMIは新しいアドレスを入手すると同時に出力しますが，これは先行するクロック・サイクルの終わりに近くなってからです．スタティック・メモリ・デバイスにインターフェースするには，アドレスをサイクルの終了後まで安定して保持しなければなりませんから，このパイプライン動作を除去しなければなりません．これは，アドレス・パイプライン・イネーブル信号である*ape*を使えば最も簡単にできます．いくつかのメモリ・デバイスが早期のアドレスに恩恵を受け，いくつかがスタティックである場合，外部ラッチを使ってスタィック・デバイスにアドレスをリタイミングするか，現在アドレスされているデバイスに合うよう*ape*を制御する必要があります．

演習9.1.1 本章で説明したプロセッサ・コアを振り返り，ARM7TDMIからARM10TDMIまでにコア性能を8倍にした基本テクニックについて考えなさい．

演習9.1.2 プロセッサへの供給電圧を設計者が自由に変更できるシステムの場合，性能（V_{dd}に比例して増減）と電力効率（$1/V_{dd2}$として増減）のトレードオフを行うことができます．したがって電源の変化の影響を計算に入れないアーキテクチャの電力効率を示す基準は，MIPS/Wです．

この基準に基づいて，本章に出てきたプロセッサ・コアを比較しなさい．

演習9.1.3 前の演習の結果を使って，なぜ低電力システムの設計者がたんにアーキテクチャとして最も効率の高いプロセッサ・コアを選択し，必要なシステム性能を出すよう電源電圧を調節してはならないのかを考えなさい．

第10章 メモリ階層

現代のマイクロプロセッサは，非常に高速で命令を実行します．この性能を完全に発揮させるには，プロセッサは非常に大容量の高速メモリ・システムを接続していなければなりません．メモリが小さすぎれば，プロセッサを常時動作させるためのプログラムを保持できませんし，メモリが遅すぎれば，メモリからの命令の供給がプロセッサの実行速度に追い付きません．

都合の悪いことに，メモリが大きくなればアクセス速度は落ちます．したがって，高性能プロセッサを十分生かせるほど大きく，かつ高速な単一のメモリを設計するのは不可能といえます．

しかし，小さな高速メモリと大きな低速主メモリを組み合わせ，一般的なプログラムの統計では，たいていの場合，一見大きな高速メモリのように動作するような複合メモリ・システムを構築することは可能です．小さな高速メモリ・コンポーネントはキャッシュと呼ばれ，プロセッサがもっとも頻繁に使う命令とデータのコピーを自動的に保存します．キャッシュの効果は，プログラムの空間的局所性と時間的局所性によって決まります．

この2レベルのメモリの概念は，多くのレベルをもつメモリ階層に拡張でき，コンピュータ・バックアップ(ディスク)ストアは階層の一部と見なすことができます．適切なメモリ管理機構があれば，プログラムのサイズはコンピュータのメイン・メモリによってではなく，それよりずっと大きなハード・ディスクのサイズによって限界が決まります．

10.1　メモリ・サイズと速度

一般的なコンピュータ・メモリ階層は，いくつかのレベルからなり，各レベルには特徴的なサイズと速度があります．

(1) プロセッサ・レジスタ

これは，メモリ階層のトップと見なすことができます．RISCプロセッサは通常，計128バイトにのぼる約32本の32ビット・レジスタをもち，アクセス時間は数nsecです．

(2) オンチップ・キャッシュ・メモリ

この容量は，8から32Kバイトで，アクセス時間は10nsec前後です．

(3) オンチップ2次キャッシュ

これは，容量数百Kバイトで，アクセス時間は数十nsecです．高性能デスクトップ・システムがもつことがあります．

メイン・メモリ

数Mバイトから数十MバイトものDRAMをもち，アクセス時間は約100nsecです.

バックアップ・ストア

通常，ハード・ディスクになりますが，数百Mバイトから数Gバイトにのぼり，アクセス時間は数十msecです.

メイン・メモリとバックアップ・ストアの性能差は，システムに2次キャッシュがない場合でさえ，どの隣接する階層よりも開きがあります.

レジスタが保持するデータは，コンパイラやアセンブリ・プログラマによって直接制御されますが，残りの階層レベルについての内容については通常自動的に管理されます.キャッシュは，アプリケーション・プログラムからは事実上不可視で，命令やデータのブロックまたは「ページ」を単位として，ハードウェア制御のもとに階層を上下に移行されます．メイン・メモリとバックアップ・ストア間のページングはOSが制御し，アプリケーション・プログラムからは，つねに透過的です.メイン・メモリとバックアップ・ストアの性能差は非常に大きいため，レベル間のデータ移行時期を決めるには，ここでさらに高度なアルゴリズムが必要です.

組み込み型システムは，通常，バックアップ・ストアをもたないので，ページングを使いません．反面，多くの組み込み型システムはキャッシュを組み込んでおり，ARMのCPUチップはさまざまなキャッシュ構成を採用しています．そこで，本章ではキャッシュ構成について少し詳しく見てみましょう.

◆メモリ・コスト

高速メモリは低速メモリより1ビット当たりが高価になりますから，低速メモリと同様のビット当たり平均コストで高速メモリに近い性能を出すこともメモリ階層化の目的です.

10.2 オンチップ・メモリ

マイクロプロセッサが最高の性能を発揮するためには，何らかの形のオンチップ・メモリが不可欠です．現在のクロック速度では，ウェイト状態ゼロのアクセス速度をサポートできるのはオンチップ・メモリだけだからです．オンチップ・メモリを使用すると，電力効率も向上し，オフチップ・メモリに比べて電磁妨害も減少します.

◆オンチップRAMの利点

多くの組み込み型システムでは，単純なオンチップRAMのほうが多くの理由でキャッシュより好まれています.

(1) オンチップRAMのほうが，単純，安価で電力消費が少なくすむ

以降の節で，キャッシュ・メモリを効果的に動作させるためには論理の点でかなりのオーバヘッドがかかることを説明します．またキャッシュ・メモリは，市販のキャッシュで適当なものがない場合，設計コストも多くかかります.

(2) オンチップRAMのほうが，決まったタイミングで動作をする

キャッシュ・メモリは複雑な動作をするため，特定の条件下でどこまでうまく動作するのか予測が困難です．とくに，割り込み応答時間を保証するのが困難なことがあります.

キャッシュと比べた場合のオンチップRAMの弱点は，プログラマによる明示的な管理が必要である点です．キャッシュは通常，プログラマに対して透過的です.

プログラム・ミックスがよく定義されており，プログラマの制御下にあれば，オンチップRAMは実質的にソフトウェア制御キャッシュとして使用可能です．ただしアプリケーション・ミックスが予測できない場合，この制御作業は非常に難しくなります．したがって，アプリケーション・ミックスが未知である汎用システムでは通常，キャッシュを使用します．

オンチップRAMの重要な長所は，プログラマが未来の処理負荷に関する知識を使って，その中のスペースを割り当てることができる点です．自ら判断するキャッシュは，過去のプログラム動作に関する知識しかありませんから，クリティカルな未来のタスクに前もって備えることは絶対にできません．クリティカルなタスクが厳しいリアルタイム制約を満たさなければならない場合，やはりこれが重要な違いになってきます．

システム設計者は，上記すべての要素を考慮して，特定のシステムはどちらが適切なアプローチかを決定しなければなりません．オンチップ・メモリを選択するときは，慎重に指定してください．オンチップ・メモリはプロセッサを常に忙しく動かすほど速く，クリティカル・ルーチンを含むことができる大きさがなければなりませんが，速すぎても（電力を消費しすぎる）大きすぎても（チップ領域をとりすぎる）いけません．

10.3　キャッシュ

初代RISCプロセッサが導入されたのは，まだ標準メモリ・チップがマイクロプロセッサより高速だった時代です．しかし，この状況は長く続きませんでした．半導体プロセス技術におけるその後の進歩はマイクロプロセッサを高速化しましたが，別の形でメモリ・チップを向上させることにもなりました．標準DRAMの場合，多少速くもなりましたが，ほとんど容量増を目的に進歩しています．

◆プロセッサおよびメモリの速度

1980年当時，一般的なDRAMチップはデータ4Kビットを保持できました．そして，1981年と1982年に，16Kビット・チップが登場しました．これらのサイクル速度は，ランダム・アクセスで3～4MHz，ローカル・アクセス（ページ・モード）ではその2倍の速度で動作しました．当時のマイクロプロセッサは，1秒に約200万回のメモリ・アクセスを要求することが可能でした．

2000年，DRAMはチップ当たり256Mビットの容量があり，30MHz前後で動作します．マイクロプロセッサは，1秒に1億回のメモリ・アクセスを要求できます．プロセッサがメモリよりかなり高速になると，キャッシュ・メモリの助けなしには潜在的な性能を発揮することはできません．

キャッシュ・メモリは小型かつ非常に高速なメモリで，最近使ったメモリ値のコピーを保持します．プログラムにはトランスペアレントに動作し，自動的にどの値を保持するか，どの値をどの値で上書きするかを決定します．最近では，通常，プロセッサと同じチップ上に実装されています．キャッシュが役立つのは，プログラムが局所性をもつからです．つまり，どんな場合でも，同じ命令を何度も（ループなど），そしてデータの同じ領域に対して（スタックなど）実行する傾向があるのです．

◆統合キャッシュとハーバード・キャッシュ

キャッシュを構築する方法は数多くあります．最上位レベルでは，プロセッサは次の二つのうちどちらかの構成をもちます．

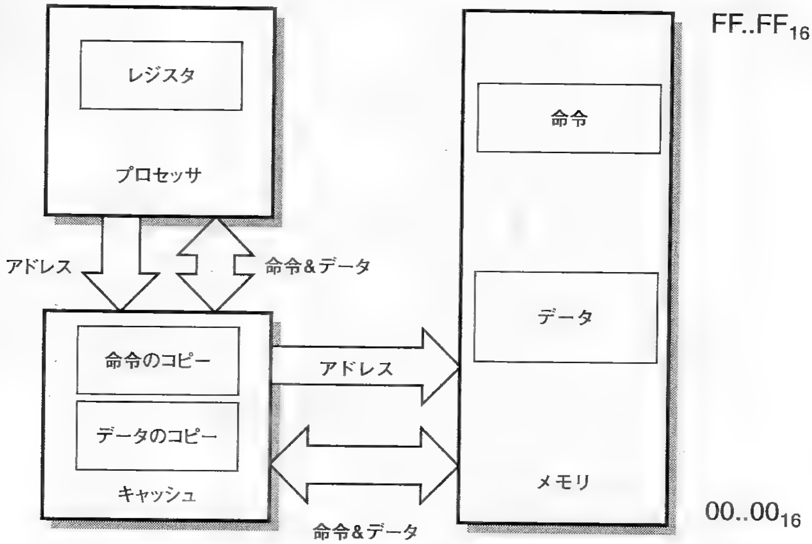
(1) 統合キャッシュ

命令とデータ両方のための単一キャッシュです（図10.1）．

◆命令，データで分かれたキャッシュ

この構成は「修正ハーバード・アーキテクチャ」と呼ばれることがあります(図10.2).

どちらの構成も長所があります．統合キャッシュでは，現在のプログラムの要求に合わせて命令が使用するキャッシュ・メモリの比率を自動調整するため，固定パーティションより高性能が得られます．一方，別々のキャッシュでは，1クロック・サイクルでロード命令とストア命令が実行できます．



〔図10.1〕命令&データ統合キャッシュ

◆キャッシュの性能測定

プロセッサが高クロック速度で動作できるのは要求するメモリ項目がキャッシュにある場合だけですから，全体的なシステム性能はキャッシュにないメモリ・アクセスの割合にかかっています．キャッシュにある項目へのアクセスは**ヒット**，キャッシュにない項目へのアクセスは**ミス**と呼びます．キャッシュに当たったメモリ・アクセスの全体比は，**ヒット率**といい，通常%で表します．キャッシュに当たらなかったほうは**ミス率**と呼びます．

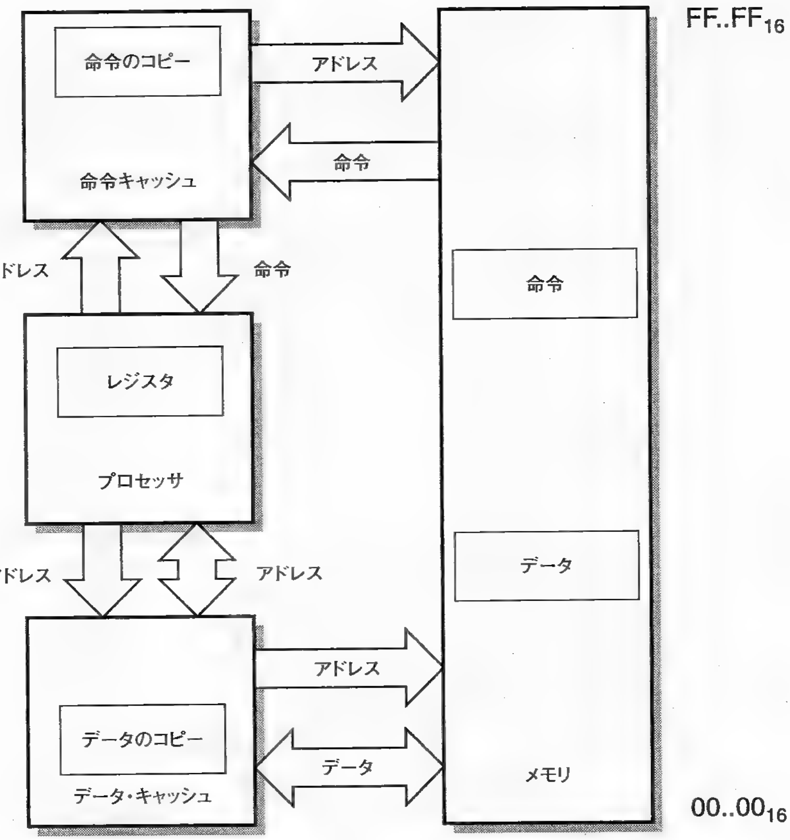
うまく設計されてキャッシュのミス率が数%なら，現代のプロセッサは潜在能力を発揮できます．ミス率は，サイズ(キャッシュ内メモリのバイト数)や構成など多くのキャッシュ・パラメータによって決まります．

◆キャッシュ構成

キャッシュが保持する主メモリからの項目は種類が動的に変化しますから，キャッシュはデータおよびデータが主メモリでストアされていたアドレスの両方を記憶する必要があります．

◆ダイレクトマップ・キャッシュ

これらのコンポーネントのうちもっとも単純な構成は，図10.3に示したダイレクトマップ(direct-mapped)キャッシュです．ダイレクトマップ・キャッシュでは，メモリ・アドレスの一部によってアドレス指定されるメモリ内に(インデクス)，1本のデータ・**ライン**(線)，アドレス・**タグ**を一緒にストアします．



〔図10.2〕命令/データ分割キャッシュ

メモリ項目がキャッシュにストアされているかどうかチェックするには，関連のアドレス・ビットを使ってキャッシュ・エントリにアクセスします．そして，最上位アドレス・ビットをストアされているタグと比較します．それらが同じであれば，求める項目はキャッシュにあるということです．同じライン中の別のアイテムにアクセスするには最下位アドレス・ビットを用います．

このもっとも単純なキャッシュ構成がもつ特徴は，複雑な構成のものとは対照的です．

(1) 各メモリ項目はキャッシュ内で独自の位置にストアされている．同じキャッシュ・アドレス・フィールドをもつ二つの項目はその位置の使用を争うことになる．

(2) タグ・フィールドにストアする必要があるのは，アドレス・ラインの１本の選択やキャッシュRAMをアドレス指定するのに使っていないビットだけである．

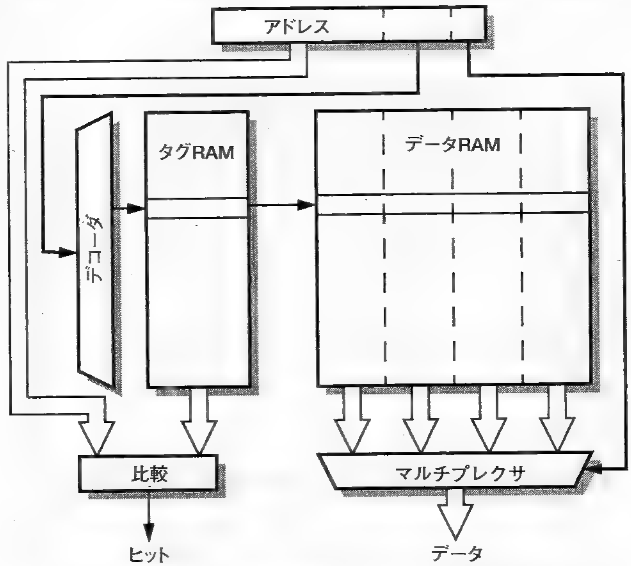
(3) タグおよびデータ・アクセスは同時に実行できるため，キャッシュ・アクセス時間はどんな構成よりも短くなる．

(4) タグRAMは，通常，データRAMよりずっと小さいのでアクセス時間が短く，データ・アクセス時間内にタグ比較が完了する．

一般的なダイレクトマップ・キャッシュは，16バイト・ラインを使う場合でデータ8Kバイトをストアします．したがって，512ラインあることになります．32ビット・アドレスでは，ライン内でバイトをアドレス指

するのに4ビット，ラインを選択するのに9ビットをもっています．ですから，残りは19ビット・タグで，タグ・ストア1Kバイト強を要求します．

キャッシュにデータをロードすると，**データ・ブロック**がメモリからフェッチされます．ブロック・サイズよりライン・サイズが小さいと意味がありません．ブロック・サイズがライン・サイズより小さいとき，タグ・ストアを拡張してライン内の各ブロック用の有効ビットを含めるようにします．ライン・サイズとブロック・サイズを同じにすると，もっとも簡単な構成ができます．



〔図10.3〕ダイレクトマップ・キャッシュ構成図

◆セット・アソシアティブ・キャッシュ

複雑になるにつれ，セット・アソシアティブ・キャッシュは，メモリ項目を複数のキャッシュにストアすることによって，競合によって起こる問題を減らします．図10.4に2ウェイ・セット・アソシアティブ・キャッシュを示しました．図からわかるとおり，この種のキャッシュは事実上，並行して動作している二つのダイレクトマップ・キャッシュです．キャッシュに与えられたアドレスのデータはどちらの半分にあることもありますから，各メモリ・アドレスは2ヵ所のうちどちらかにストアされます．ダイレクトマップ・キャッシュで一つの位置について競合している2項目は，どちらかの場所を占めることができ，キャッシュは両方でヒットすることになります．

16バイト・ラインをもつ8Kバイト・キャッシュは，2分したキャッシュの両方に256ラインをもつことになりますから，32ビット・アドレスの4ビットでラインからバイトを選択し，8ビットがキャッシュの両方から1ラインずつ選択します．したがってアドレス・タグは20ビットで1ビット長くなります．アクセス時間はダイレクトマップ・キャッシュの場合より少しだけ長くなりますが，それは2分した両方からのデータをマルチプレックスする必要があるからです．

新しいデータ項目をキャッシュに置く際，それをどちらの半分に置くか決定しなければなりません．方法はいくつかありますが，もっとも一般的なものは次のとおりです．

(1) ランダム割り当て

ランダムまたは疑似ランダム値に基づいて決定します.

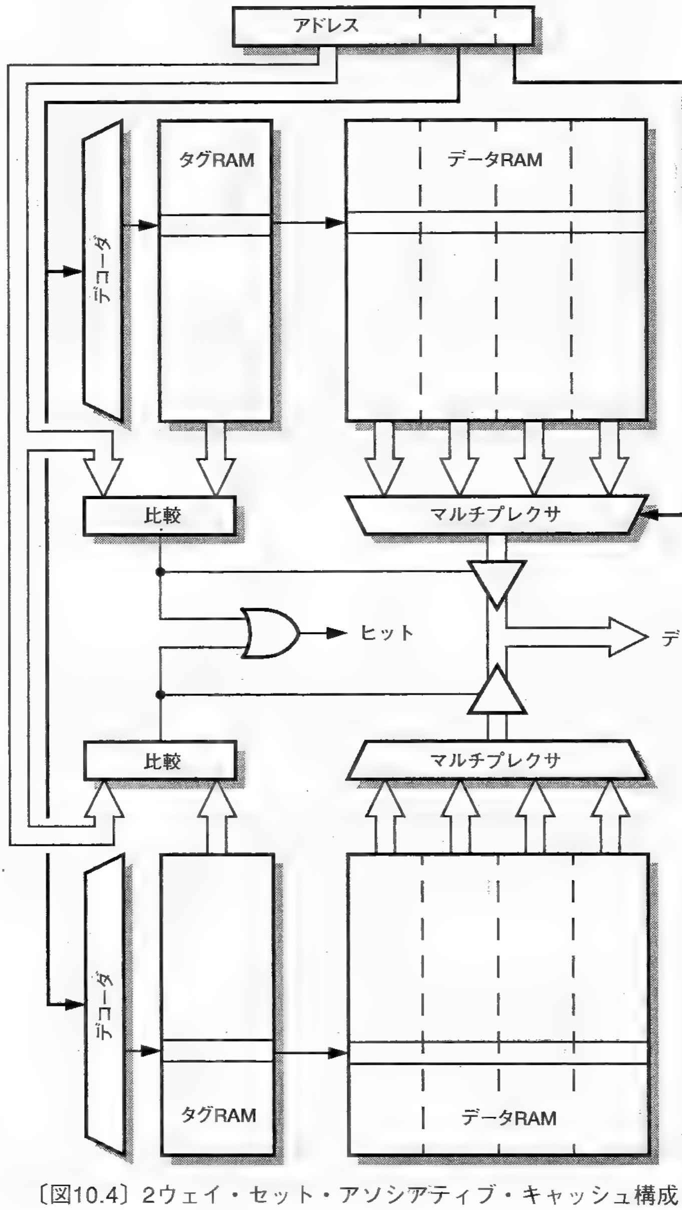
(2) 最低使用頻度(LRU)

キャッシュは最後にどちらの半分にアクセスしたかを覚えており,新しいデータをもう一方に割り当てます.

(3) ラウンドロビン(巡回とも呼ばれる)

キャッシュは最後にどちらの半分に割り当てたかを覚えており,新しいデータをもう一方に割り当てます.

セット・アソシアティブ方式は,2ウェイ以上どこまででも連想度(アソシアティブ)を拡張していくことが

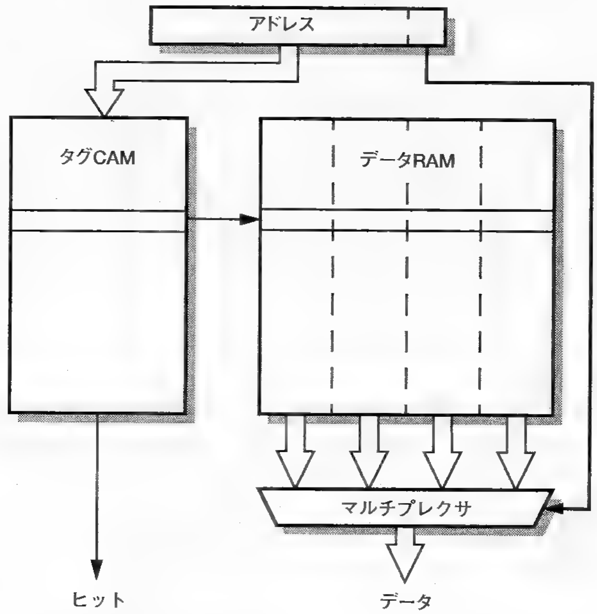


〔図10.4〕2ウェイ・セット・アソシアティブ・キャッシュ構成

できますが，実際には4ウェイ以上にすると複雑化に対する割が合わなくなります．

◆フルアソシアティブ・キャッシュ

連想方式でのもう一つの方法に，フルアソシアティブ・キャッシュがあり，VLSIテクノロジ社のツールで設計が可能です．ダイレクトマップ・キャッシュをどんどん小さいコンポーネントに分割していくのではなく，アドレス付き内容メモリ（CAM）を用いて個別にタグ・ストアを設計しています．CAMセルは内蔵コンパレータをもつRAMセルですから，タグ・ストアに基づいたCAMは並列サーチを実行してどの位置にあるアドレスでも見つけることができます．フルアソシアティブ・キャッシュの構成は，図10.5のとおりです．



〔図10.5〕フルアソシアティブ・キャッシュ構成

キャッシュ内のデータ位置に暗黙のアドレス・ビットはありませんから，タグはライン内でバイト・アドレスに用いたアドレス・ビット以外，すべてのアドレス・ビットをストアしなければなりません．

◆書き込み方法

これまで述べてきた方式は，明らかに読み出しアクセスの方法です．新しい読み出しアドレスを与えられると，キャッシュは自分がアドレスされたデータをもっているかどうかチェックします．もっていれば，データを供給します．もっていなければ，主メモリからデータ・ブロックをフェッチしてキャッシュの適切な位置にストアし，要求されたデータをプロセッサに与えます．

プロセッサが書き込みサイクルを実行するには，他にも方法があります．一般的に用いられている書き込み方法を複雑でない順に並べると次のようになります．

(1) ライトスルー

すべての書き込み動作は主メモリに渡されます．アドレス指定した位置がキャッシュ内にあれば，キャッシュを更

新して新しい値を保持させます．プロセッサは書き込み動作中，主メモリの速度まで速度を落とします．

(2) ライト・バッファ付きライトスルー

この場合も，すべての書き込み動作は主メモリに渡し，キャッシュが適切にアップデートしますが，プロセッサを主メモリ速度にまで落とす代わりに，書き込みアドレスおよびデータをライト・バッファにストアし，ライト・バッファが書き込み情報を高速に受け取ります．その後，プロセッサが次のタスクを処理する間に，ライト・バッファが主メモリ速度でデータを主メモリに転送します．

(3) コピーバック（ライトバックとも呼ばれる）

コピーバック・キャッシュは，主メモリと一貫性が保たれていません．書き込み動作はキャッシュのみアップデートするため，キャッシュ・ラインはいつ変更されたのかを覚えておかなければなりません（通常は各ラインまたはブロックの**ダーティ・ビット**を使用する）．ダーティ・キャッシュ・ラインが新しいデータに割り当てられた場合は，ラインが再使用される前に必ずメモリへコピーバックします．

ライトスルー・キャッシュは実装がもっとも単純で，メモリが最新に保たれるという利点があります．欠点は，書き込み転送のたびにプロセッサは，メモリ速度のボトルネックに陥ることです．ライト・バッファを追加すると，書き込みトラフィックが外部書き込み帯域幅を超えるまでプロセッサが動作を継続できます．コピーバック・キャッシュは必要な外部書き込み帯域幅を低減します．なぜなら，最終値がメモリに再書き込みされるまでに同じ位置が何度も書かれるからです．しかし，実装はかなり複雑で，一貫性の欠如を管理するのが困難です．

◆キャッシュの特徴まとめ

キャッシュの構成を定義するさまざまなパラメータを表10.1にまとめました．最初は，キャッシュとメモリ管理ユニット（MMU）の関係で，詳しくは「仮想キャッシュと物理キャッシュ」で説明します．他はこの節ですでに述べました．

〔表10.1〕キャッシュ構成選択肢のまとめ

キャッシュ構成	選択肢		
キャッシュ-MMU関係	物理キャッシュ	仮想キャッシュ	
キャッシュ内容	単一命令&データ・キャッシュ	別々の命令&データ・キャッシュ	
連想度	ダイレクトマップ RAM-RAM	セット・アソシアティブ RAM-RAM	フルアソシアティブ CAM-RAM
リプレースメント・ストラテジ	巡回	ランダム	LRU
書き込みストラテジ	ライトスルー	ライト・バッファ付き ライトスルー	コピーバック

10.4　キャッシュ設計の例

10.3節で述べたように，キャッシュの構成を選択するには，キャッシュの大きさ，連想度の程度，ラインと

ブロックのサイズ，置換アルゴリズム，書き込み方法など，いくつかの要素を考慮する必要があります．これらの選択がキャッシュの性能に与える効果を分析するには，詳しいアーキテクチャ・シミュレーションが必要です．

◆ARM3キャッシュ

1989年に設計されたARM3は，オンチップ・キャッシュを組み込んだ最初のARMチップであり，これらのパラメータが性能とバス使用に与える影響について詳しい研究がなされました．この研究では，ARM2で複数のベンチマーク・プログラムを実行しつつアドレス・トレースをキャプチャするため，特別に設計したハードウェアを使用しました．そして，これらのトレースを分析して多様な構成の動作をモデル化するには，ソフトウェアを使用しました(現在，デスクトップ機はハードウェア・サポートなしで十分に大型のプログラムをシミュレートするだけの性能を持っているので，特別なハードウェアは不要)．

研究は，キャッシュから得られると期待される性能向上の上限を設定することから始まりました．要求されるデータをつねに含んでいる「完璧な」キャッシュがこの上限設定のモデルとなりました．実際のキャッシュはときどきミスを起こしますから，いつもヒットするものに比べて性能がいいはずはありません．

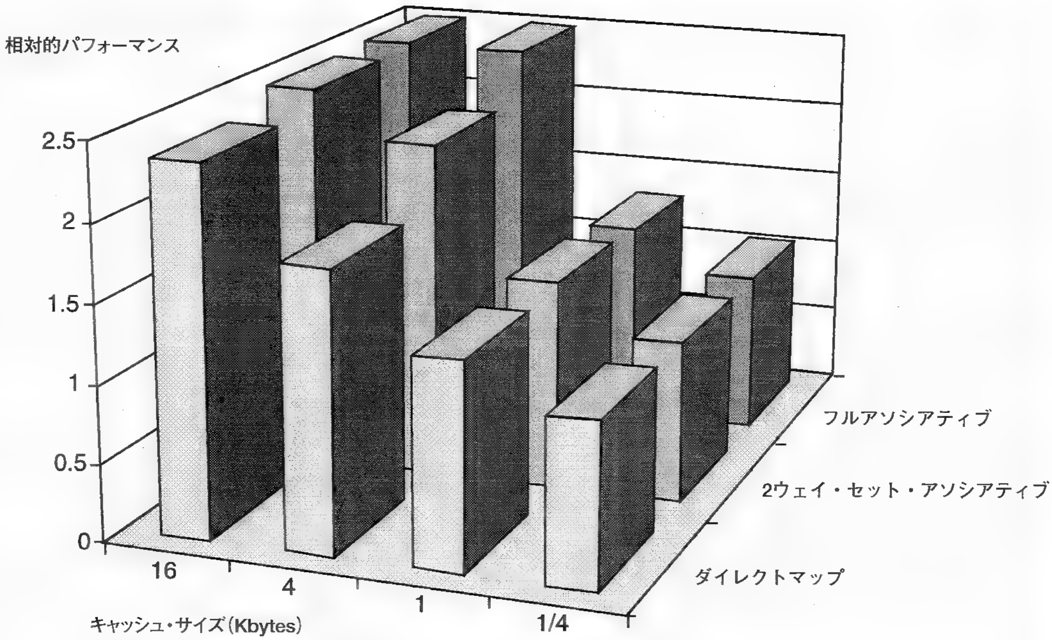
キャッシュ・スピードと外部メモリ・スピード(それぞれ20MHzと8MHz)については現実的な仮定を使って，3種類の完璧なキャッシュがモデリングされました．つまり，命令だけを含むキャッシュ，命令とデータの混合を含むキャッシュ，データだけを含むキャッシュです．結果は表10.2のとおりで，キャッシュのないシステムの性能を基準としています．この結果から，キャッシュに保持するうえで命令がもっとも重要な値であることがわかりますが，データ値も保持するとさらに25%性能が向上しています．

〔表10.2〕「完璧な」キャッシュの性能

キャッシュ形式	実行比較
ノー・キャッシュ	1
命令オンリ・キャッシュ	1.95
命令&データ・キャッシュ	2.5
データ・オンリ・キャッシュ	1.13

前もって，キャッシュ書き込み方法はライトスルーと決定していましたが(おもに単純であるという理由で)，キャッシュが書き込みミスを検出したら書き込みアドレスからデータ・ラインをロードすることも可能です．この「書き込みミスの時点で割り当てる」方法は，短期間研究されましたが，複雑性が大幅に増す割には利益が少ししかないことがわかったので，まもなく廃棄されました．そして問題は，読み出しミスで割り当てを行う統合された命令&データ・キャッシュにとって最適な構成で，チップ面積と電力的制約に見合ったものを見つけることだけになりました．

さまざまなキャッシュ構成とサイズが研究され，結果は図10.6のようになりました．もっとも単純なキャッシュ構成はダイレクトマップ・キャッシュですが，16 Kバイトのサイズにしてもこのキャッシュは「完璧な」キャッシュより大幅に性能が劣っています．次に単純なのは，2ウェイ・セット・アソシアティブ・キャッシュです．この場合，16 Kバイトのサイズで性能は「完璧な」キャッシュに比べて1%未満程度しか違いません．



〔図10.6〕サイズと構成に対する統一キャッシュの性能

しかし，ARM3の設計当時(1989年)，16 Kバイトのキャッシュは大きなチップ面積を必要とし，4 Kバイトのキャッシュはあまり性能がよくありませんでした(使用したアドレス・トレースの生成に使うプログラムによって結果は大きく違ってくるが，上記は一般的な結果である)．

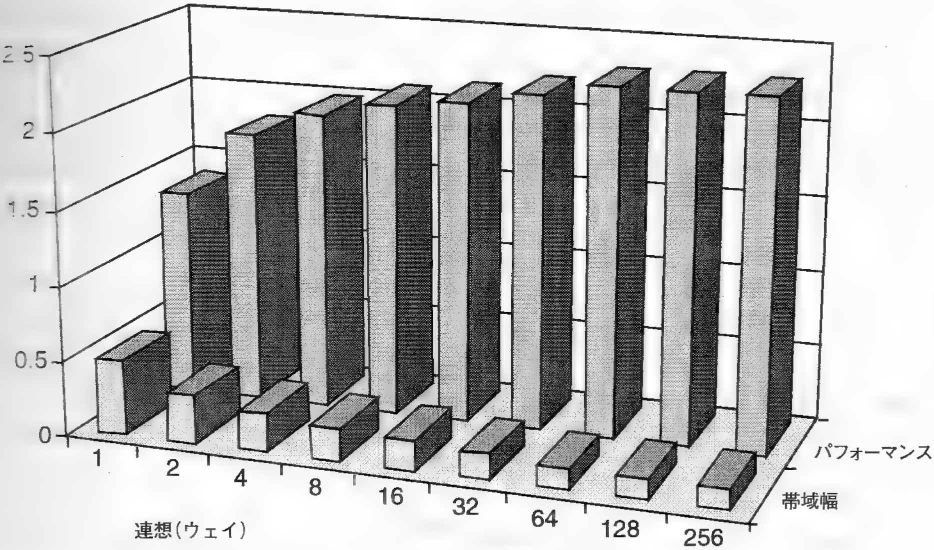
逆にもっとも複雑な構成でみると，フルアソシアティブ・キャッシュは，サイズが小さくても大幅に性能がよく，テストに使用したベンチマーク・プログラムでは「完璧な」性能を発揮しました．ここでは置換アルゴリズムはランダムです．LRUでも非常に似た結果が出ます．

このあと，キャッシュ・モデルは4ワード・ラインを使用するよう変更されました．これは，タグ・ストアにかかる面積コストを減らすために必要だったのです．性能には，ほとんど影響がありませんでした．

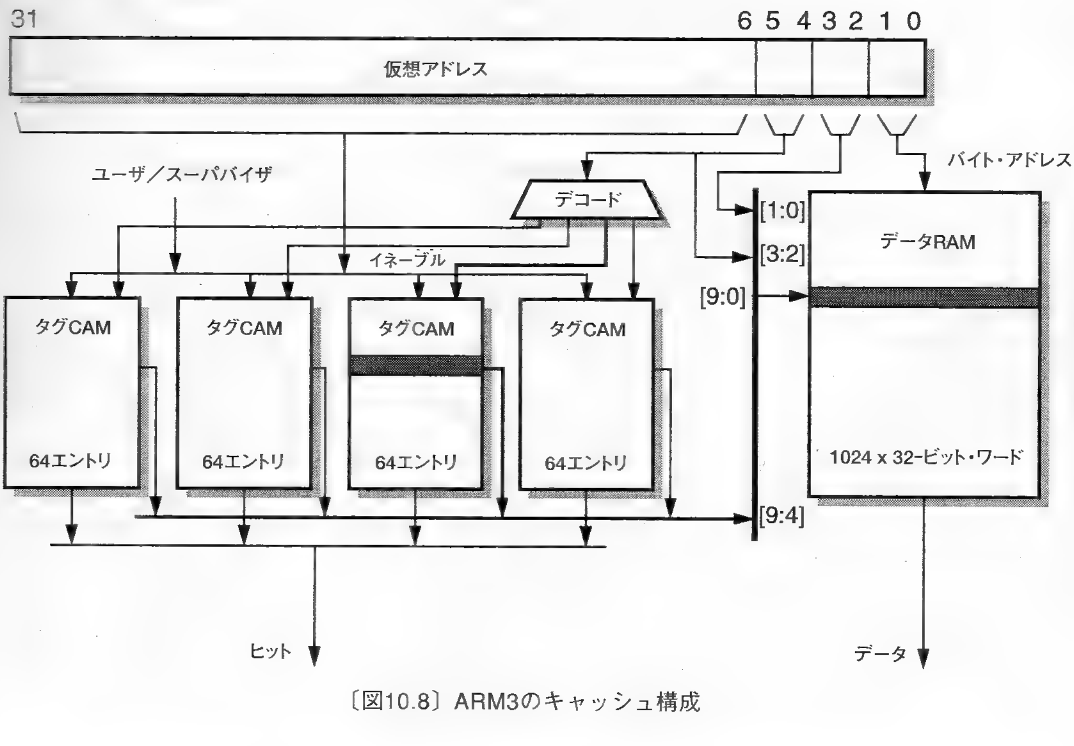
フルアソシアティブ・キャッシュには，大きなCAM(アドレス付き内容メモリ)タグ・ストアが必要ですが，これは4ワード・ラインを使っても通常かなりの電力を消費します．消費電力は，CAMを小さいコンポーネントに分化することによって低減できますが，こうすると連想度が低下します．4Kバイトキャッシュを使った場合における連想度とシステム性能の関係分析は，図10.7のとおりです．この図は，フルアソシアティブ(256ウェイ)からダイレクトマップ（1ウェイ）まで全連想度のシステム性能を示しています．ダイレクトマップから2ウェイセット・アソシアティブまでの性能向上がもっとも大きいものの，64ウェイ・アソシアティブまで一貫して顕著な性能向上が見られます．

したがって64ウェイ・アソシアティブCAM-RAMキャッシュは，256 CAMエントリを4分割して電力を節約することができるとともに，フルアソシアティブ・キャッシュと同じ性能をもつと考えられます．各レベルの連想度について必要な外部メモリ帯域幅も図10.7に示してあります(キャッシュなしプロセッサに対する相対値)．もっとも高性能なものが最も少ない外部帯域幅を必要としていることに注意してください．外部アクセスは内部動作に比べて多くのエネルギを使いますから，キャッシュは性能を向上すると同時にシステムに必要な電力を減らしていることになります．

キャッシュの構成は，図10.8のとおりです．仮想アドレスの最下位2ビットは32ビット・ワード内の1バイトを選択，次の2ビットはキャッシュ・ライン内の1ワードを選択，次の2ビットは四つの64エントリCAMタグ・ストアの一つを選択します．残りの仮想アドレスは，選択したタグ・ストア（残りのタグ・ストアは電力



〔図10.7〕性能/メモリ帯域幅と連想度の効果



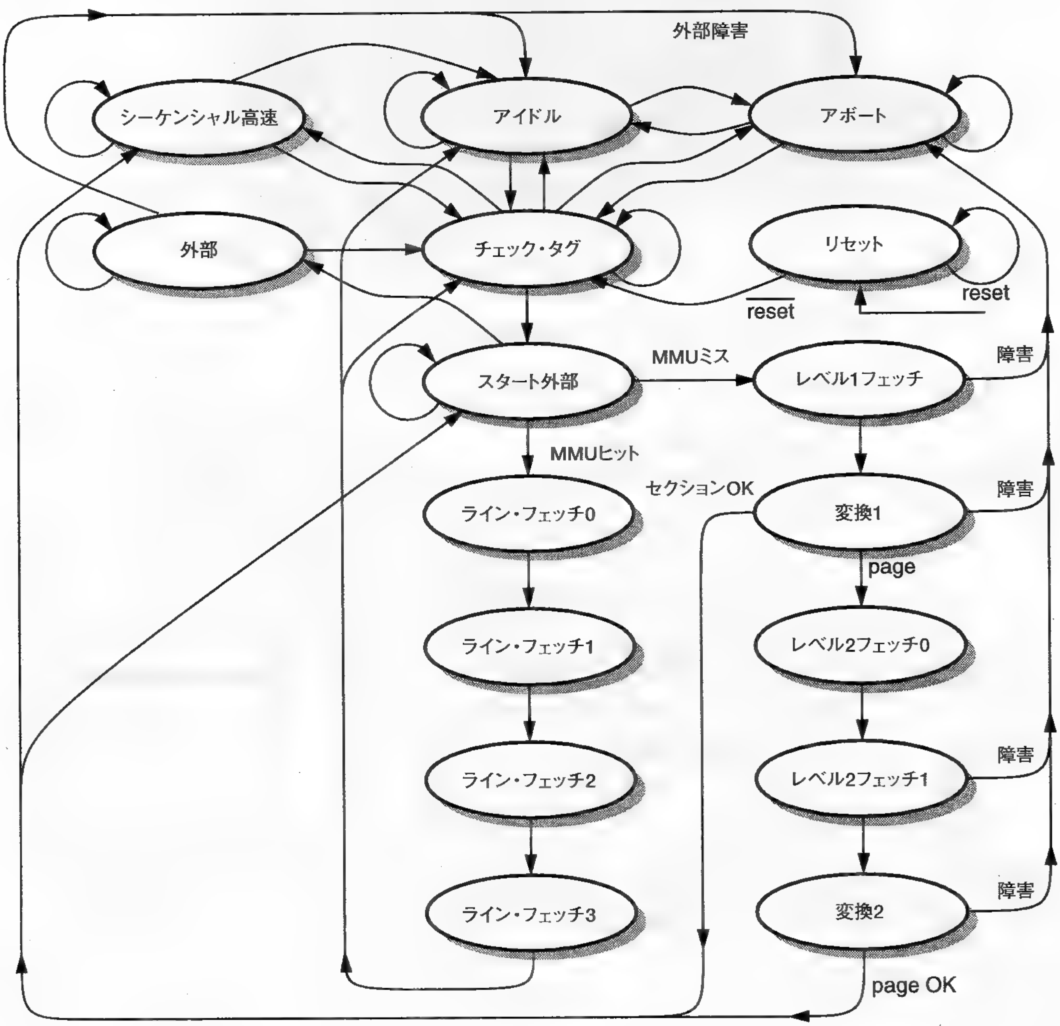
〔図10.8〕ARM3のキャッシュ構成

節約のためにディセーブルされている)に入り，データがキャッシュ内にあるかどうかのチェックに使います．結果はキャッシュ・データRAM中のデータ・アドレスにヒット，またはミスのどちらかです．

◆ARM600のキャッシュ制御FSM

キャッシュを管理するのに必要な種類の制御論理を示すために，ARM600のキャッシュ制御有限ステート・マシンを下記に説明します．ARM600キャッシュは，その設計を10.4節で説明したARM3から借用しており，やはり10.5節で述べたのと同様な方式の変換システムをもっています．

ARM600は，2本のクロックにしたがって動作します．高速クロックは，キャッシュで動作したりライト・バッファに書き込んだりする際にプロセッサ・サイクル時間を定義します．メモリ・クロックは，プロセッサが外部メモリにアクセスする際のスピードを定義します．コアに与えられるクロックは，この2種類のクロッ



〔図10.9〕ARM600キャッシュ制御ステートマシン

ク・ソースの間を動的に切り替わります．各クロックは互いに非同期でもかまいません．メモリ・クロックが高速クロックの単なる分周クロックであってもかまいませんが，もしそうであれば，同期オーバヘッドを避けるようプロセッサを構成することが可能です．

通常，プロセッサは高速クロックを使ってキャッシュから動作します．キャッシュ・ミスが起こると（またはキャッシュ不可能なメモリへの参照が起こると），プロセッサはメモリ・クロックに同期し，一つの外部アクセスかキャッシュ・ラインフィルを実行します．2クロックの間を切り替わる場合，同期にオーバヘッドがかかりますから（準安定性のリスクを許容可能なレベルに抑えるため），プロセッサは高速クロックに戻るかどうか決定する前に次のアドレスをチェックします．

この動作を制御する有限ステート・マシンは，図10.9のとおりです．初期化のあと，プロセッサは高速クロックで動作しながらチェック・タグ状態に入ります．アドレス指定されたデータがキャッシュにあるかどうかによって，プロセッサは以下のいずれかの方向に進むことができます．

(1) アドレスがノン・シーケンシャルである限り，MMU内でフォールトせず，キャッシュにある読み出しかバッファ付き書き込みとなります．ステート・マシンはチェック・タグ状態のままとなり，データ値は毎クロック・サイクルで戻されるか書き込まれます．

(2) 次のアドレスが，同じキャッシュ・ラインにあるシーケンシャル読み出しか，シーケンシャルなバッファ書き込みの場合，ステート・マシンはシーケンシャル高速状態に移行します．この状態では，タグをチェックせず，かつMMUをアクティブにせずにデータにアクセスすることができます．これにより電力を節約し，プロセッサ・コアからの*seq*信号を利用します．ここでも毎クロック・サイクルで，データ値の書き込みまたは読み出しが行われます．

(3) アドレスがキャッシュ内にないか，バッファなし書き込みの場合，外部アクセスが必要です．これは，スタート外部で開始します．外部状態では，キャッシュ不可能なメモリからの読み出しとバッファなし書き込みは，一つのメモリ・トランザクションとして完了します．キャッシュ可能な読み出しでは，必要な変換情報がすでにMMUにない場合はフェッチしてから，4ワード・ライン・フェッチを実行します．

(4) プロセッサがメモリを使用しないサイクルは，アイドル状態で実行します．

変換プロセスのいくつかの時点で，アクセスが完了できないことが明らかとなり，アボート状態に入ることがあります．キャッシュ不可能な読み出しおよびバッファなし書き込みも，外部ハードウェアによってアボートされることがあります．

10.5 メモリ管理

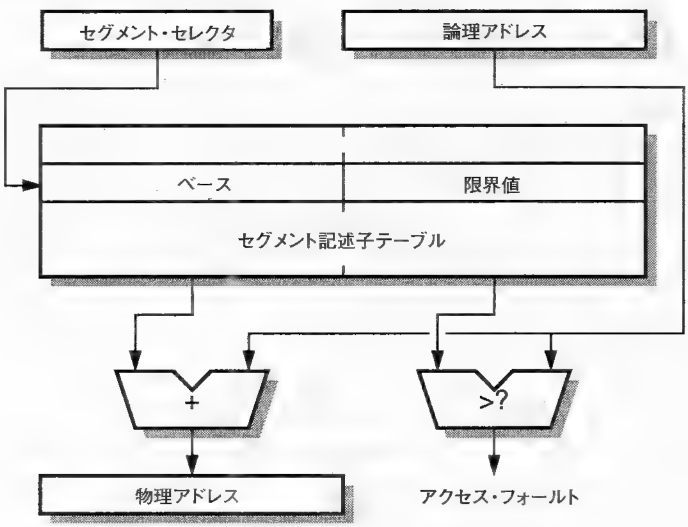
現代のコンピュータ・システムでは，多くのプログラムが同時にアクティブになっています．もちろん一つのプロセッサは，一度に一つのプログラムからの命令しか実行できませんが，アクティブなプログラム間を素早くスイッチすることによって，全部同時に実行しているように，少なくとも人間の時間感覚では見えるのです．

高速スイッチを管理するのは，OSですから，アプリケーション・プログラマはあたかもプログラムがマシン全体を所有しているかのようにプログラムを書くことができます．このように感じさせるメカニズムは**メモリ管理ユニット**（MMU）と呼ばれます．メモリ管理には，**セグメント**，**ページ**という二つの主要なアプローチがあります．

◆セグメント

もっとも単純なメモリ管理は，メモリをある種類の情報を含んだセグメント集合としてアプリケーションに認識させることです．たとえば，プログラムには，全命令を含んだコード・セグメント，データ・セグメント，スタック・セグメントがあります．各メモリ・アクセスは，MMUに対してセグメント選択子と論理アドレスを与えます．各セグメントはベース・アドレスとそれに関連する限界があります．論理アドレスはセグメント・ベース・アドレスからのオフセットで，限界値より大きくはなりません．さもなければアクセス違反が起こって例外を引き起こします．セグメントには他のアクセス制御があることもあります．たとえば，コード・セグメントが読み出し専用で書き込もうとすると例外を起こすこともあります．

セグメント化したMMUのアクセス・メカニズムを図10.10に示します．



〔図10.10〕セグメント化メモリ管理方式

セグメント化により，同時に実行する個々のプログラムにとってメモリは自分専用の私的な空間に見え，同じメモリ空間を他のプログラムとトランスペアレントに共存することができるようになります．しかし，共存するプログラムが変化したり，使用可能なメモリが限られている場合は問題が出てきます．セグメントのサイズはさまざまなので，時間がたつにつれて新たに使用可能な空きメモリ空間は断片化し，新しいプログラムが起動できない場合が出てきます．これは，空きメモリが容量的に足りないのではなく，空きメモリがすべて小さな断片になっていて，どれも新しいプログラムが要求するセグメント・サイズに足りないからなのです．

この問題は，OSがセグメントを動かして空きメモリを合体させ，大きな塊にすることで軽減できます．しかし，これでは不十分なため，現在の多くのプロセッサでは，**ページ**と呼ばれる固定サイズのメモリの塊に基づいたメモリ・マッピング方式を組み込んでいます．セグメント化とページングの両方を含むアーキテクチャもありますが，ARMなど多くはセグメント化なしでページングのみをサポートしています．

◆ページング

ページング・メモリ管理方式では，論理アドレスと物理アドレスの両方ともがページと呼ばれる固定サイズ

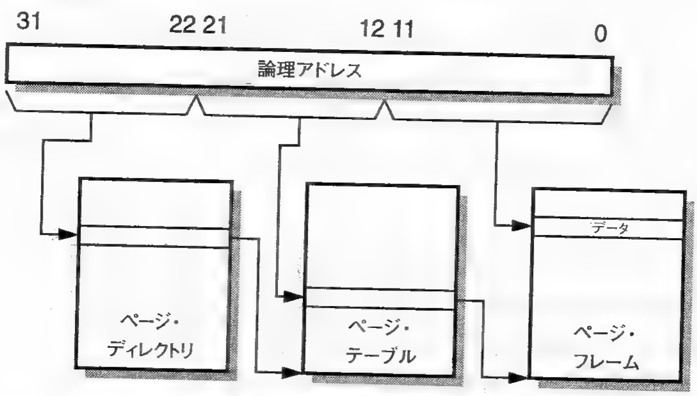
の単位に分割されています．1ページは通常，数Kバイト・サイズですが，ページ・サイズはアーキテクチャによって異なります．論理ページと物理ページの関係は，主メモリの**ページ・テーブル**にストアしてあります．

この関係を一つのテーブルにして変換するには，非常に大きなテーブルが必要になることは，単純な足し算でわかります．1ページが4Kバイトなら，32ビット・アドレスのうち20ビットを変換する必要があります．これにはテーブル上のデータ$2^{20}\times 20$ビット，つまりは最低2.5Mバイトのテーブルが必要になり，小さなシステムにおいてはオーバヘッドが大きくなりすぎます．

代わりに，多くのページング・システムでは二つ以上のレベルのテーブルを使用します．たとえばアドレスの上位10ビットは，第1レベル・ページ・ディレクトリの中で適切な第2レベル・ページ・テーブルを見つけます．また，アドレスの次の10ビットは物理ページ番号を含むページ・テーブル・エントリを特定します．この変換方式は，図10.11に示してあります．

大切なのは，ここであげた具体的数字で32ビットを各ディレクトリとページ・テーブル・エントリに割り当てるとすると，ディレクトリと各ページ・テーブルはちょうど4Kバイト，つまり1メモリ・ページずつ占めることになります．小さなシステムにおける最小オーバヘッドはページ・ディレクトリに4Kバイトと1ページ・テーブルに4Kバイトです．これは，4Mバイトまでの物理メモリを管理するには十分です．32Gバイト・メモリは，4Mバイトのページ・テーブルを必要とするでしょうが，これだけのメモリがあればオーバヘッドもおそらく許容範囲になると思われます．

11.6節で説明するARM MMUは，ここで示したものとは多少異なるビット割り当てを用いますが(より大きいメモリ・ブロックの単一階層変換もサポートしている)，原理は同じです．



〔図10.11〕ページング・メモリ管理方式

◆仮想メモリ

どちらのメモリ管理方式でも，セグメントやページがアクセスされると，つねに「不在」とマークし，例外を起こすようにすることが可能です．その場合，割り当てるメモリを使い果たしたOSは，ページやセグメントを主メモリからバックアップ・ストアへ(通常はハード・ディスクだが)アプリケーションからはわからないように移動し，「不在」とマークします．すると，物理メモリを別用途に割り当てることが可能になります．プログラムが「不在」ページやセグメントにアクセスしようとすると例外が起こり，OSがページやセグメントを主メモリに戻してから，プログラムがアクセスを再試行します．

ページング・メモリ管理方式と一緒に実装されたとき，このプロセスは**要求時ページング仮想メモリ**(demand-paged virtual memory)と呼ばれます．プログラムを，動作するコンピュータ上で使える物理

メモリより大きな仮想メモリ空間を占めるように書くことができるのです．なぜならOSは，プログラムやデータの小さな一部を必要に応じて取ってくることができるからです．一般的プログラムは，一部のコードを非常に頻繁に使い，また一部はほとんど使いません．つまり，めったに使わないルーチンをディスクに移動しても，性能にはほとんど影響しないということです．しかし，この機能を使いすぎると，OSはページを入れたり出したり高速でスイッチしなければなりません．これは**スラッシング**と呼ばれ，反対に性能を低下させます．

◆再起動可能命令

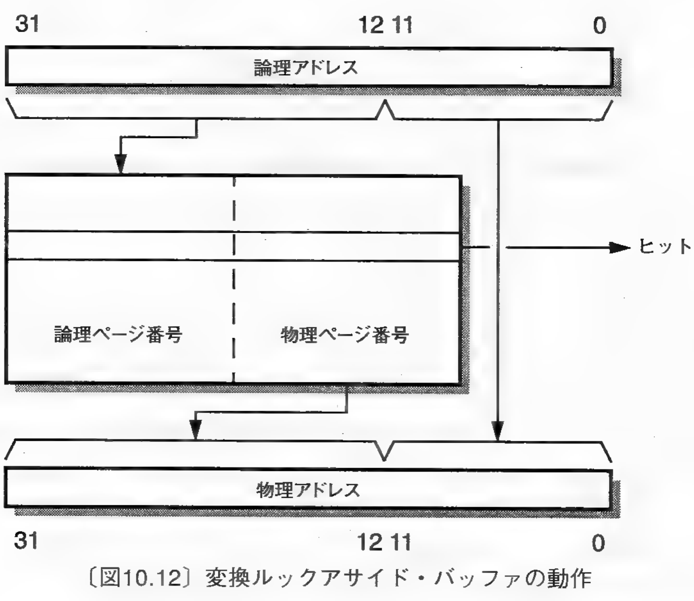
仮想メモリ・システムにおける重要な条件は，メモリ・アクセス障害を起こしうる命令に際して，要求されるメモリをOSがページ・インし，あたかも障害がなかったかのようにプロセッサが元のプログラムを再開できる状態にしておくことです．これはメモリにアクセスする命令をすべて**再起動可能**(restartable)にすることで可能になります．プロセッサは状態を十分保存し，OSがレジスタ値を回復できるようにしておき，それによってページが主メモリにきたとき，ページが最初からあったのと同じ結果が得られるように失敗した命令を再試行できるようにします．

高性能かつハードウェア冗長性を最低に保ちながらこの条件を満たすのは，プロセッサ設計において通常もっとも難しいことです．

◆変換ルックアサイド・バッファ

上記のページング方式によって，プログラマはメモリ使用に関して完全な自由と透過性が得られます．しかし，データ自体にアクセスする前に各メモリ・アクセスにつきページ・ディレクトリとページ・テーブル二つの追加メモリ・アクセスのオーバヘッドがかかるため，性能が非常に落ちるように思えるかもしれません．

このオーバヘッドは，最近使ったページ変換をキャッシュする変換ルックアサイド・バッファ(**TLB**：translation look-aside buffer)の実装で通常回避できます．命令やデータ・キャッシュと同様(10.3節のとおり)，連想度の程度と交換方法に関連した構成上のオプションがあるのです．通常，ラインおよびブロック・サイズ



〔図10.12〕変換ルックアサイド・バッファの動作

はページ・テーブル・エントリひとつと同じで，一般的なTLBサイズは64エントリ前後で，データ・キャッシュよりかなり小さくなります．プログラムの局所性によってこのサイズのTLBを使って1%前後のミス率を達成できます．ミスには，2回の追加メモリ・アクセスだけテーブル・ウォーキング・オーバヘッドがかかります．

TLBの動作は図10.12のとおりです．

◆仮想キャッシュと物理キャッシュ

システムにMMUとキャッシュの両方が組み込まれているとき，キャッシュは仮想(MMU前)アドレスまたは物理(MMU後)アドレスのいずれかで動作します．

プロセッサがアドレスを生成すると同時にキャッシュ・アクセスを開始でき，データがキャッシュにあればMMUを起動する必要もない点で仮想キャッシュは有利です．欠点は，キャッシュが**同義語**(synonyms)，つまりキャッシュ内に同じ主メモリ・データ・アイテムのコピーをもつ場合です．同義語が起こるのは，アドレス変換メカニズムが一般的にオーバラップ変換を許すからです．プロセッサが一つのアドレス・ルートを通じてデータ・アイテムを変更すると，キャッシュは2番目のコピーを更新できませんから，キャッシュ内に矛盾を引き起こします．

物理メモリ・アドレスは，独自のデータ・アイテムと対応しているため，物理キャッシュでは同義語の問題は起こりません．しかし，MMUが毎キャッシュ・アクセスでアクティブになり，キャッシュ・アクセス開始前にMMUがアドレス変換を完了しなければならないようなMMUおよびキャッシュ構成もあるため，キャッシュ遅延時間が長くなります．

うまくシーケンシャル・アクセス・コストを回避する物理キャッシュ配置では，ページングMMUが高位アドレス・ビットのみに影響する一方，キャッシュにアクセスするのは低位アドレス・ビットのみという事実を利用しています．これらがオーバラップしないとすれば，キャッシュとMMUアクセスが並行して進行できます．MMUからの物理アドレスはキャッシュからの物理アドレスタグと比較するのにちょうどいい時間に到着するため，アドレス変換時間はキャッシュ・タグ・アクセスの後ろに隠れてしまいます．この最適化はフルアソシアティブ・キャッシュでは適用できませんし，MMUが使うページ・サイズがキャッシュの直接アドレス部分より大きい場合のみ使えます．たとえば4Kバイト・ページの場合，ダイレクト・マップ・キャッシュの最高サイズは4Kバイト，2ウェイ・セット・アソシアティブ・キャッシュは8Kバイトなどと制限します．

実際には，仮想キャッシュ，物理キャッシュの両方が商用化されており，前者は同義語問題をおさえるためにソフトウェア規則を用い，後者は上記の最適化を利用するか性能コストを許容するか，ということになります．

10.6 例題と演習

＜例題10.1＞ 1Kバイト・ページのシステムにおいての4ウェイ物理キャッシュはどこまで大きくできるか？

本章の「仮想キャッシュと物理キャッシュ」に説明したとおり，TLBとキャッシュ・アクセスを並行して実行するとします．キャッシュの各セクションは，最高1Kバイトになりますから，総キャッシュ・サイズは最高4Kバイトになります．

演習10.1.1 ライン・サイズが16バイトのとき，このキャッシュ中のタグ・ストアはどれだけのメモリを要求

しますか？

演習10.1.2 上記の例で，TLB，データ・キャッシュ・タグ，データ・メモリの面積比率を見積りなさい．

<例題10.2> TLBがすべての物理ページの変換を含むには，どのぐらいの大きさになるか？

4Kバイト・ページでは，1Mバイト・メモリは256ページですから，TLBには256エントリ必要です．TLBはもはや自動キャッシュである必要はありません．TLBミスはページが物理メモリに不在という意味ですから，ディスク転送が必要になります．TLBをソフトウェアによって維持するオーバヘッドはディスク転送のコストに比べればささいなものです．

全物理メモリを含むTLBは「反転ページ・テーブル」の形をしています．ちょうどそのような変換方式はAcorn Archimedes機に使用していた初期のARMメモリ・コントローラ・チップでも使われていました．図10.12を見ると，変換ハードウェアはCAMでもかまいません．物理ページ番号ストアは単純なハード・ワイヤード・エンコーダで，CAMは各物理ページに1エントリをもちます．

Acornメモリ・コントローラ・チップには128エントリをもつCAMがあり，システム内の物理メモリ量にしたがってページ・サイズが決まっていました．1Mバイト・システムは8Kバイト・ページ，4Mバイト・システムは32Kバイト・ページをもっています．4Mバイト以上に拡張するには，ページ・サイズを32Kバイトのままにし，第2メモリ・コントローラを追加します．CAMはソフトウェアによって維持されたため，複雑なテーブル・ウォーキング・ハードウェアは必要ありませんでした．変換テーブルはすべてソフトウェアによって定義されていました．

演習10.2.1 64エントリTLBと比較した128エントリ反転ページ・テーブルのダイ面積を見積りなさい．CAM1ビットはRAM1ビットの2倍の面積を必要とするとします．

第11章 OSのアーキテクチャ・サポート

OSの役割は，複数のプログラムが妨害し合うことなく同時に動作する環境を作り，かつ安全なデータ共有をサポートすることです．OSは，またハードウェア機能に対するクリーンなインターフェースも提供します．

プロセス間の干渉は，メモリ管理と，各プロセスが自分のメモリ領域にのみアクセスできるようにする保護方式によって最小化することができます．各プロセスは，システム・メモリに対して独自の視点を与えられ，プロセスがスイッチするとメモリの見え方は新しいプロセスのものへと動的に変換し，前のプロセスが使っていたメモリはすべて見えなくなります．これを効率的に動作させるには，高度なハードウェア・サポートが必要です．

データ共有は，保護方式に穴を開けることでもあるため，細心の注意を払って制御します．共有構造に対する無統制なアクセスは，非常にわかりにくいプログラム誤動作につながるので，統制された方法をとらなければなりません．

ハードウェア機能へのアクセスには，低レベルのビット操作も含みます．これは各プロセスが独立して行うのではなく，通常OSが中心となって詳細を処理します．そのうえで，プロセスは，システム・コールをとおして高レベルの入出力機能にアクセスします．

ARMアーキテクチャは，OSのこれらの全側面をサポートするための特別な機能を組み込んでいます．

11.1 OSの基礎

OSの役割は，マシンのハードウェア資源と，マシン上で動作するアプリケーション・プログラムとの間の統一されたクリーンなインターフェースを提供することです．もっとも高度なOSとは，複数の異なるユーザが同時に複数の汎用プログラムを実行させるための機能を備えたものです．

◆マルチユーザ・システム

マルチユーザ・システムで，同じマシン上の各プログラムが他プログラムの存在をいちいち考慮しなければならないとすると非常に不便です．同時に動作しているプログラムの数やタイプは未知ですし，動作ごとに異なってくるからです．そのためマルチユーザOSは，各プログラムに完全な**仮想マシン**を用意します．各プログラムは，あたかもそれ一つだけが動作しているかのように記述されます．他プログラムの存在を感じさせる影響は唯一，プログラム動作が遅くなることだけです．

マシン上に複数のプログラムが同時に存在しうるにも関わらず，プロセッサはたった1セットのレジスタし

かもっていません（ここでマルチプロセッサ・システムは考慮しない）から，つねに1プログラムしか実行できません．一見，同時に動作しているように見せるには**タイムスライス**（time-slicing）で可能です．つまり各プログラムがプロセッサ内で交互に動作しているからです．プロセッサは，人間の感覚からすると非常な高速で動作しています．そのため，各プログラムがたとえば1秒の間に各数回も動作しながら，全プログラムは実行していくのです．OSは**スケジューリング**を行い（どのプログラムがいつ動作するかを決定する），各プログラムにCPU時間を均等に与えたり，特定のプログラムに**優先度**（priority）を与えたりします．

プログラムがプロセッサから切り離されるのは，

①OSがタイマ割り込みに呼ばれ，そのプログラムに十分時間をかけたと決定したとき

②プログラムが低速周辺アクセス（ディスク・アクセスなど）を要求して応答があるまで処理ができないとき

のどちらかです．プロセッサ内でプログラムをアイドル状態にしておくよりは，OSは動作中のプログラムを切り替えて，時間を有効に活用できる別のプログラムをスケジュールします．

◆メモリ管理

プログラムが動作する仮想マシンを作るために，OSはプログラムがしかるべきメモリ位置でコードやデータにアクセスできる環境を構築しなければなりません．あるプログラムが使うアドレスが別のプログラムの使うアドレスにぶつからないようにするため，OSはメモリ変換を使って，プログラム・コードやデータをロードした**物理アドレス**を適切な**論理アドレス**でプログラムに渡します．プログラムは，OSが管理する論理・物理アドレス変換メカニズムをとおしてメモリを見ます．

◆保護

複数のユーザが同じマシンでプログラムを動作しているとき，1ユーザのプログラムで発生したエラーが他のユーザのプログラムを妨害しないようにすることが重要になります．また，他のプログラムを妨害しようという悪意のある試みからも保護する必要があります．

各プログラムに仮想マシンを与えているメモリ・マッピング・ハードウェアは，あるプログラムが使用するメモリを他のプログラムには隠ぺいすることによって保護を可能にしています．しかし，これは行き過ぎになると非効率的です．なぜなら，有用な関数ライブラリなどを含むメモリの共有領域があれば，マシン全体としてのメモリ使用が節約できるからです．これらの領域を読み出し専用や実行専用にして1プログラムが共有するコードを破壊することができないようにすれば，問題は解決します．

悪意のあるユーザが他人に被害を与える方法の一つに，OS状態のふりをして変換テーブルを変更することによってメモリ管理システムの保護を乗り越える方法があります．多くのシステムは，アクセス制御された特権システム・モードをもち，このモードからしか変換テーブルにアクセスできないようにすることで，この問題に対処しています．

ずるがしこい人間の悪意のある妨害にも安全なコンピュータ・システムを設計するのは難しい問題で，アーキテクチャ・サポートも多少必要になります．ARMでは，アクセス制御された特権プロセッサ・モードと，メモリ管理ユニット内の多様なメモリ保護があります．しかし，ARMが悪意のあるユーザに対する保護が必要なシステムで使われることがほとんどないため，これらの機能はたいていの場合，不慮のプログラミング・エラーを探知してソフトウェアをデバッグするために使用されています．

◆資源割り当て

同時に動作している二つのプログラムがシステム資源（resource）に対して競合する要求を出すことがあり

ます．たとえば，一つのプログラムがディスクの一部からデータを要求するとします．ディスク・ドライブがデータを探している間，そのプログラムはスイッチ・アウトされてしまい，代わりにスイッチ・インしたプログラムがすぐにディスクの別部分からデータを要求することがあります．ディスク・ドライブがこれらの要求に直接応答すると，プログラムは交互に制御を行い，ディスクは二つが探すものの間で揺れ動いてどちらのデータ領域も見つける時間がもてません．そして，システムはディスク・ドライブが消耗するまでデッドロック状態になります．

この種の問題を防ぐため，すべての入出力要求はOSを通ります．OSは最初のプログラムの要求を受け入れた後，2番目のプログラムの要求をキューに待たせ，最初のプログラムの要求が終わってから処理します．

◆単一ユーザ・システム

一人のユーザがシステムを使用しているときも，同時に数プログラムが動作すると上記のほとんどが当てはまります．同じマシンを共有している悪意的なユーザの恐れはありませんが，各プログラムが自分の空間で動作し，1プログラムのエラーが他のプログラムにエラーを起こさないようにすることは有効です．悪意のあるユーザの心配がなくなったことにより，プログラムがシステム特権をとれなくするようにする必要がなくなります．これが意図せず起こることはほとんどありません．

しかし，単一ユーザのように見えるデスクトップ機もコンピュータ・ネットワークに接続することが多くなり，他ユーザによってリモートでプログラムを動作できるようになりました．そのようなマシンは明らかにマルチユーザと見なし，適切なレベルの保護を組み込むべきです．

◆組み込み型システム

組み込み型システムは，上で説明した単一ユーザ，マルチユーザ汎用システムとはかなり異なります．これは通常，固定したプログラムがセットされて動作し，新しいプログラムを導入するメカニズムはもっていません．したがって，悪意のあるユーザの問題はおそらくありません．

OSは，アクティブな各プログラムにクリーンな仮想マシンを与え，他プログラムのエラーから保護し，CPU使用時間をスケジュールするなど，ほぼ同様の役割を果たします．

多くの組み込み型システムは，スケジュールの優先順位を決定していかなければならないようなリアルタイム制約のなかで動作するからです．コスト面から，汎用マシンによくあるOSの使用も除外します．それらは大きなメモリ・リソースを要求する傾向があるからです．このことから，リアルタイムOS(**RTOS**)が開発されました．これは，数Kバイトのメモリを使用する組み込み型システムに必要なスケジューリングとハードウェア・インターフェース機能をもちます．

より小型の組み込み型システムは，このコストにさえ耐えられず，「OS」がまったくいらないほど単純なスケジューリング条件(一つの固定プログラムがつねに動作するなど)の場合もあります．その場合，単純な「モニタ」で入出力機能へのクリーンなインターフェースなどわずかなシステム機能を果たすだけで十分です．そのようなシステムでは，ほとんどの場合メモリ管理ハードウェアは不要であり、プロセッサの論理アドレスを使って直接メモリにアクセスします．組み込み型システムにキャッシュ・メモリがある場合，メモリのどの領域がキャッシュ可能かを定義する何らかのメカニズムが必要です(I/O領域はキャッシュできないため)が，これは完全なメモリ管理システムよりずっとシンプルなものになります．

◆本章の構成

メモリ管理の一般的な原理は，前の章で説明しました．本章の次節では，ARMシステム制御コプロセッサ

とそれが制御するメモリ管理システムを紹介し，アドレス変換を備えた完全なMMUや，アドレス変換を必要としない組み込み型システム向けの単純な「保護ユニット」にも触れます．

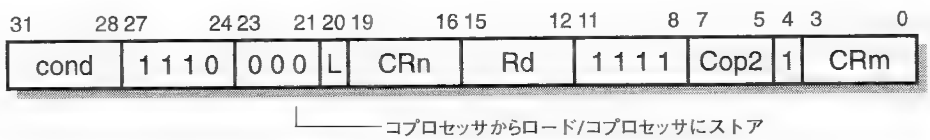
その後の節では，同期，コンテクスト・スイッチ，割り込みを含む入出力デバイスの処理などOSに関する重要な問題について説明します．

11.2　ARMシステム制御コプロセッサ

ARMシステム制御コプロセッサはオンチップ・コプロセッサで，論理コプロセッサ番号15(`CP15`)を使ってオンチップ・キャッシュなどのキャッシュ，メモリ管理ユニットや保護ユニット，ライト・バッファ，プリフェッチ・バッファ，分岐ターゲット・キャッシュ，システム・コンフィグレーション信号などの動作を制御します．

◆CP15命令

制御は，`CP15`レジスタの読み出しと書き込みを通して実行されます．レジスタはすべて32ビット長で，スーパバイザ・モードで実行する`MRC`と`MCR`命令によってアクセス制限されます(5.19節を参照)．他のコプロセッサ命令やユーザ・モードでのアクセスは未定義命令トラップを起こします．これらの命令形式は図11.1のとおりです．多くの場合，`CRm`と`Cop2`フィールドは使用されずゼロでなければなりませんが，特定の命令で使用されることがあります．



〔図11.1〕CP15レジスタ転送命令

◆保護ユニット

決まったまたは制御されたアプリケーション・プログラムをもつ組み込み型システムで使用するARM CPUには，アドレス変換機能をもつ完全なメモリ管理ユニットは不要です．そのようなシステムには，もっと単純な保護ユニットで十分です．ARM保護ユニットのCP15レジスタ構成は11.3節，保護ユニットの動作については11.4節で説明します．

保護ユニットを使用するCPUは，12.1節で説明するARM740T，12.4節で説明するARM940Tです．

◆MMU

汎用アプリケーション向けARM CPUは，設計時にアプリケーション・プログラムの範囲や数がわからないので，通常，アドレス変換付きのメモリ管理ユニットを必要とします．ARM MMUのCP15レジスタ構成は11.5節で，MMUの動作については11.6節で説明します．

完全なMMUを使用するCPUは，第12章で説明するすべてのCPUです．

11.3 CP15保護ユニット・レジスタ

保護ユニット・レジスタ構造は，表11.1のとおりです．これらのレジスタは，図11.1に挙げたCP15命令を使って読み出し，書き込みを行います．このときCRnがアクセスすべきレジスタを指定します．

詳しいレジスタ機能は以下のとおりです．

〔表11.1〕
CP15保護ユニットのレジスタ構造

レジスタ	目的
0	IDレジスタ
1	コンフィグレーション
2	キャッシュ制御
3	ライト・バッファ制御
5	アクセス許可
6	領域ベース＆サイズ
7	キャッシュ操作
9	キャッシュ・ロック・ダウン
15	テスト
4，8，10-14	未使用

(1) レジスタ0(読み出し専用)

デバイス識別情報を返します．

31 … 24	23 … 16	15 … 4	3 … 0
インプリメンタ	アーキテクチャ	パーツ番号(BCD)	レビジョン

ビット[3:0]はレビジョン，[15:4]は2進化10進法で表した3桁のパーツ番号，[23:16]はアーキテクチャ・バージョン(0はversion3，1はversion4，2はversion4T，4はversion5T)，[31:24]はインプリメンタ(実装者)の商標を示すASCIIコード(ASCII 'A'=41_{16}はARM Limited，'D'=44_{16}はDECなど)を含みます．

CPUの中には，上のレジスタ0形式と少し異なるものもあり，最近のCPUでは2番目のレジスタ0をもっていて(MRC命令のCop2フィールドを変更することによってアクセスする)，キャッシュ構成の詳細を示しています．

(2) レジスタ1(読み出し/書き込み用)

システム機能と制御システムのパラメータをイネーブルにする制御情報ビットを数ビットもちます．

31	30	29 28	27	26 25	24	23 … 14	13	12	11 … 8	7	6 … 4	3	2	1	0
iA	nf	Bnk	F	Lck	S	0000000000	V	I	0000	B	000	W	C	0	M

全ビットはリセットでクリアされます．続けてセットする場合，Mは保護ユニットをイネーブル，Cはデータまたは統合キャッシュをイネーブル，Wはライト・バッファをイネーブル，Bはバイト順序をリトル・エン

ディアンからビッグ・エンディアンへスイッチ，Iは命令キャッシュとデータ・キャッシュが分かれている場合に命令キャッシュをイネーブル，Vは例外ベクタをアドレス空間の最上位近くへ移動，S，Lck，F，Bnkはキャッシュを制御(ARM740T)，nf，iAはさまざまなクロック・メカニズムを制御(ARM940T)します．

すべての実装に全ビットがあるわけではないことに注意してください．

(3) レジスタ2(読み出し/書き込み用)

八つの各保護領域についてキャッシュ可能性を制御します．

31–8	7	6	5	4	3	2	1	0
000000000000000000000000	7	6	5	4	3	2	1	0

ビット0は領域0内でロードに使うキャッシュをイネーブルし，ビット1は同じことを領域1内で行います(以下同様)．ARM940Tでは，命令ポートとデータ・ポートに別々の保護ユニットがあり，どのユニットにアクセスするかはCop2(図11.1参照)を使って決定します．つまりCop2＝0のときはデータ・ポートの保護ユニットにアクセス，Cop2＝1のときは命令ポートの保護ユニットにアクセスします．

(4) レジスタ3(読み出し/書き込み用)

各保護領域についてライト・バッファを使用すべきかどうかを定義します．形式はレジスタ2と同じですが，ARM940T命令ポートは読み出し専用ですからライト・バッファはデータ・ポートについてしかイネーブルできず，したがってCop2がつねにゼロになる点に注意してください．

(5) レジスタ5(読み出し/書き込み用)

各保護領域についてアクセス許可を定義します．

31–16	15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
0000000000000000	ap7	ap6	ap5	ap4	ap3	ap2	ap1	ap0

アクセス許可には，アクセスなし(00)，特権モード専用(01)，特権フル・アクセスおよびユーザ読み出し専用(10)，フル・アクセス(11)があります．ARM940Tでは，やはりCop2フィールドを使って命令(1)保護ユニットとデータ(0)保護ユニットを区別します．

(6) レジスタ6(読み出し/書き込み用)

八つの各領域の開始アドレスとサイズを定義します．

31–12	11–6	5–1	0
領域ベース・アドレス	000000	サイズ	E

領域のベース・アドレスは，サイズの倍数でなければなりません．サイズ・フィールドのエンコーディングは表11.2のとおりです．Eは領域をイネーブルします．

特定の領域は，CRmフィールド(図11.1参照)を0～7にセットして指定します．ARM940Tのようなハーバード・コアでは，命令メモリ・ポートとデータ・メモリ・ポートに別々の領域レジスタがあり，レジスタ2で述べたようにCop2がどちらのメモリ・ポートをアドレスするかを指定します．

(7) レジスタ7

これは，さまざまなキャッシュ動作を制御しますが，その動作はARM740TとARM940Tでは異なります．

■レジスタ9

これは，ARM940Tで使用し，キャッシュ領域をロック・ダウンします(ARM740Tでは，この目的にレジスタ1の中の特定ビットを使用).

■レジスタ15

これは，ARM940Tで使用し，キャッシュ割り当てアルゴリズムをランダムからラウンドロビンに変更します．これは，シリコンの製品テスト中だけ使用するものです．

11.4 ARM保護ユニット

組み込み型アプリケーション用のARMCPUにはメモリ保護ユニットが組み込まれており，多様な保護機能やキャッシュ機能をメモリの各領域について定義することによって，たとえばI/O領域を特権アクセス専用に制限してキャッシュ不可にするなどといったことができます．保護ユニットはアドレスを変換しません．アドレス変換を必要とするシステムは，11.6節で述べる完全なメモリ管理ユニットを使用しなければなりません．保護ユニットを組み込んだCPUには，ARM740T，ARM940Tがあります．

◆保護ユニットの構造

保護ユニットは，ARMの4Gバイトのアドレス空間を八つの領域に分割します．各領域にはプログラム可能な開始アドレスがあり，サイズもプログラム可能，保護およびキャッシュ特性もプログラム可能です．領域はオーバラップすることもあります．固定優先順位方式が，複数の領域に属する一つのアドレスに適用される性質を定義しています．

◆領域定義

八つの領域の開始アドレスとサイズは，CP15のレジスタ6に書き込むことによって定義します．このレジスタの形式は，前のページで定義しました．最小領域サイズは4Kバイトで，最大は4Gバイトです．CP15のレジスタ6に書き込まれた*Rd[5:1]*はサイズを定義しますが，これは表11.2に示した最大値と最小値の間で2バイトの任意の累乗にセットすることができます．開始アドレスは選択したサイズの倍数で，*Rd[31:12]*で定

〔表11.2〕
保護ユニットの領域サイズ

Rd[5:1]	領域サイズ
01011	4Kバイト
01100	8Kバイト
01101	16Kバイト
01110	32Kバイト
…	…
11100	512Mバイト
11101	1Gバイト
11110	2Gバイト
11111	4Gバイト

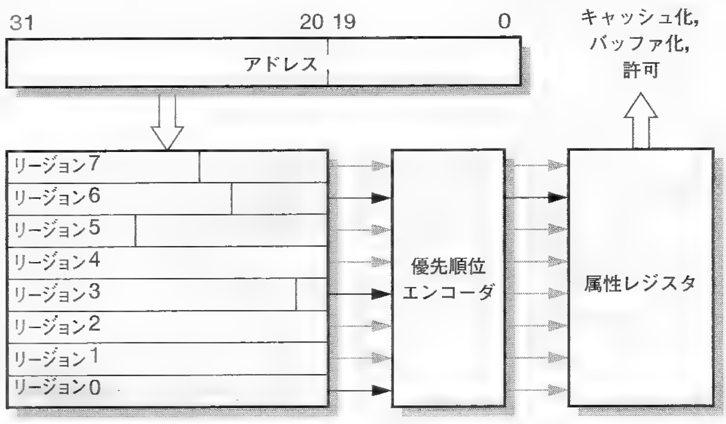
義します．各領域は，*Rd[0]*でイネーブルされないかぎり保護機能が動作しません．

◆領域の優先順位

領域は，オーバラップするようにプログラムすることが正式に認められています．複数の領域に属するアドレスが保護ユニットに提示されると，固定優先順位方式によってどの領域がその属性を定義するかが決定されます．領域7が最高の優先順位，領域0が最低の優先順位をもち，あとの領域もその番号順に優先順位をもちます．

サイズの倍数でなければならないというレジスタ開始に関する制約により，アドレスが領域にあるかどうかをチェックするには，アドレスの最上位数ビット(20ビット以下)を領域開始アドレスの対応ビットと比較するだけとなります．それらのビットがマッチすればアドレスが領域内にあることになり，マッチしなければ領域外にあることになります．加算も減算も必要ないので，この処理は高速です．したがって保護ユニットの構成は，図11.2のようになります．この図では，領域0が4Gバイトのアドレス空間を全部カバーしているので，全アドレスがその中にあります．領域1，2，4は4Kバイト，領域3，6，7，5はサイズが増加しています．例に挙げたアドレスは，領域0，3，6(黒い矢印で示したとおり)に属していますから，優先順位エンコーダがもっとも高い優先順位をもつ領域6の属性を選択します．

アドレスがイネーブルされた領域のどれにも当てはまらない場合，保護ユニットはアボートを生成します．



〔図11.2〕
保護ユニットの構成図

◆ハーバード・コア

ハーバード・アーキテクチャを用いるARMコア(ARM940Tなど)は，命令ポートとデータ・ポートに別々の保護ユニットがありますから，計16領域あることになります．

11.5 CP15 MMUレジスタ

MMUレジスタ構造は，表11.3のとおりです．これらのレジスタは，図11.1に挙げたCP15命令を使って読み出し，書き込みを行います．このときCRnがアクセスすべきレジスタを指定します．

詳しいレジスタ機能は後で述べるとおりです．

〔表11.3〕 CP15 MMUレジスタ構造

レジスタ	目的
0	IDレジスタ
1	制御
2	変換テーブル・ベース
3	ドメイン・アクセス制御
5	フォールト・ステータス
6	フォールト・アドレス
7	キャッシュ操作
8	TLB操作
9	読み出しバッファ操作
10	TLBロックダウン
13	プロセスIDマッピング
14	デバッグ用
15	テスト&クロック制御
4，11-12	未使用

(1) レジスタ0(読み出し専用)

識別情報を返します．

31–24	23–16	15–4	3–0
インプリメンタ	0000000A	パーツ番号(BCD)	レビジョン

ビット[3:0]はレビジョン，[15:4]は2進化10進法で表した3桁のパーツ番号，[23:16]はアーキテクチャ・バージョン('A'=0はversion3，'A'=1はversion4)，[31:24]はインプリメンタ(実装者)の商標を示すASCIIコード(ASCII 'A'=41_{16}はARM社，'D'=44_{16}はDEC社など)を含みます．

CPUの中には，上のレジスタ0形式と少し異なるものもあり，最近のCPUでは2番目のレジスタ0をもっていて(MRC命令のCop2フィールドを変更することによってアクセスする)，キャッシュ構成の詳細を示しています．

(2) レジスタ1(アーキテクチャversion3では書き込み専用，version4では読み出し/書き込み用)

システム機能と制御システムのパラメータをイネーブルにする制御情報ビットを数ビットもちます．

31–15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00000000000000000	RR	V	I	Z	F	R	S	B	L	D	P	W	C	A	M

全ビットはリセットでクリアされます．続けてセットする場合，Mは MMUをイネーブル，Aはアドレス・アラインメント・フォールト・チェックのイネーブル，Cはデータまたは統合キャッシュをイネーブル，Wはライト・バッファをイネーブル，Pは例外処理を26ビットから32ビットへスイッチ，Dはアドレス範囲を26

ビットから32ビットへスイッチ，Lはレイト・アボート・タイミングへスイッチ，Bはバイト順序をリトル・エンディアンからビッグ・エンディアンへスイッチ，SとRはMMUシステムとROM保護状態の変更，Fは外部コプロセッサ通信の速度制御，Zは分岐予測をイネーブル，Iは命令キャッシュとデータ・キャッシュが分かれている場合に命令キャッシュをイネーブル，Vは例外ベクタ・ベースを0x00000000から0xffff0000へ移動，RRはキャッシュ置換アルゴリズム(擬似ランダムまたはラウンドロビン)を制御します．すべての実装に全ビットがあるわけではないことに注意してください．

ビット[31:15]は，読み出し時には予測不能であり，リード・モディファイ・ライト・アクセスを使って保護しなければなりません．ビット[31:30]は，ARM920およびARM940でクロック制御機能などに使用します．

(3) レジスタ2(version3では書き込み専用，version4では読み出し/書き込み用)

これは，現在アクティブな第1レベル変換テーブルの開始アドレスを格納しています．16Kバイト境界で整列していなければなりません．

31 … 14	13 … 0
変換テーブル・ベース・アドレス	00000000000000

(4) レジスタ3(version3では書き込み専用，version4では読み出し/書き込み用)

これは，16個の2ビット・フィールドをもち，各フィールドが16ドメインのうち一つについてアクセス許可を指定します．詳しくは，268ページの「ドメイン」を参照してください．

31 30	29 28	27 26	25 24	23 22	21 20	19 18	17 16	15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

(5) レジスタ5(version4では読み出し/書き込み用，version3では読み出し専用で書き込みを行うとTLB全体をフラッシュする)

これは，フォールトのタイプとアボートした最後のデータ・アクセスのドメインを示します．Dはデータ・ブレークポイントでセットされます．

31 … 10	9	8	7 … 4	3 … 0
0000000000000000000000	D	0	ドメイン	ステータス

(6) レジスタ6(version4では読み出し/書き込み用，version3では読み出し専用で書き込みを行うと特定のTLBエントリをフラッシュする)

これは，アボートした最後のデータ・アクセスのアドレスを含みます．

31 … 0
フォールト・アドレス

(7) レジスタ7(version4では読み出し/書き込み用，version3では書き込み専用で単にキャッシュをフラッシュする)

これは，多数のキャッシュ，ライト・バッファ，プリフェッチ・バッファ，分岐ターゲット・キャッシュのクリーンおよび(または)フラッシュ動作を実行します．供給されるデータはゼロまたは関連の仮想アドレスです．レジスタ7へのアクセスには`Cop2`および`CRm`フィールドを用い，具体的動作を指定します．利用可能な機能は実装によって異なります．

(8) レジスタ8(version4では読み出し/書き込み用，version3では使用不可)

これは，多数のTLB動作を実行します．TLBの単一エントリまたは全体をフラッシュし，統合TLBまたは命令とデータに分かれたTLBをサポートします．

(8) レジスタ9

これは，読み出しバッファが存在すれば，それを制御します．キャッシュ・ロックダウン機能を制御するのに使うCPUもあります．

(9) レジスタ10

これは，TLBロックダウン機能がサポートされていれば，それを制御します．

(10) レジスタ13

これは，プロセスIDレジスタを通じて仮想アドレスをリマップします．このメカニズムはWindows CEをサポートするために用いられ，ARM720T，ARM920T，SA-1100など特定のCPUにのみ存在します．バーチャル・アドレスのビット`[31:25]`がゼロの場合，このレジスタのビット`[31:25]`で置き換えます．

31　　　25	24　　　　　　　　　　　　0
プロセスID	0000000000000000000000000

(11) レジスタ14

これは，デバッグ・サポートに使用します．

(12) レジスタ15

これは，テストと，一部のCPUではクロック制御目的に使用します．

11.6　ARM MMUアーキテクチャ

MMUには主な機能が二つあります．

(1) 仮想メモリを物理メモリに変換

(2) 不正アクセス・アボートによるメモリ・アクセス許可制御

ARMのMMUは，テーブル・ウォーキング・ハードウェア，最近使用したページ変換をストアするTLBをもつ「2レベル・ページ・テーブル」を使用します．プロセッサが別個に命令キャッシュとデータ・キャッシュをもつ場合は，たいてい命令TLBとデータTLBを別々にもちます．

◆メモリ粒度

メモリ・マッピングは，同じ基本メカニズムに基づきさまざまな粒度で実行されます．使用するユニットは次のとおりです．

(1) セクション

1Mバイトのメモリ・ブロックです．

(2) ラージ・ページ

64Kバイトのメモリ・ブロックで，ラージ・ページ内では16Kバイトの各サブページにアクセス制御が適用されます．

(3) スモール・ページ

4Kバイトのメモリ・ブロックで，スモール・ページ内では1Kバイトの各サブページにアクセス制御が適用されます．

(3) タイニー・ページ

最新CPUの一部は，1Kバイトの「タイニー・ページ」もサポートしています．

通常の粒度は4Kバイト・スモール・ページです．ラージ・ページとセクションは，一つのTLBエントリで大きなデータ領域をマッピングするためにあります．大きなデータ領域を無理に小さなページにマッピングすると，状況によってはTLBの効率が悪くなります．

◆ドメイン

ドメインは，ARMのMMUアーキテクチャの変わった機能の一つです．**ドメイン**とは，特定のアクセス許可をもつセクションおよび(または)ページのグループです．これにより，多数の異なるプロセスが，互いに一定の保護を残しながら，同じ変換テーブルで動作することができます．各プロセスが独自の変換テーブルをもつよりも，かなり簡潔なプロセッサ・スイッチ・メカニズムになります．

アクセス制御は2種類のプログラムに基づいています．

(1) クライアント

ドメインのユーザを指し，ドメインを構成する各セクションやページのアクセス許可に従わなければなりません．

(2) マネージャ

ドメインのコントローラを指し，各セクションやページのアクセス許可をバイパスできます．

プログラムは，つねに，いくつかのドメインのクライアントであり，また別のドメインのマネージャでもあり，また残りのドメインには何のアクセスもありません．これは，`CP15`のレジスタ3によって制御されます．`CP15`レジスタ3は，16ドメインそれぞれに対して2ビットずつを格納し，各ドメインに関する現在のプログラム状態を示します．各2ビットの解釈は表11.4のとおりです．全ドメインに対するプログラムの関係は，`CP15`レジスタ3に新しい値を一つ書き込むことで変更できます．

◆変換プロセス

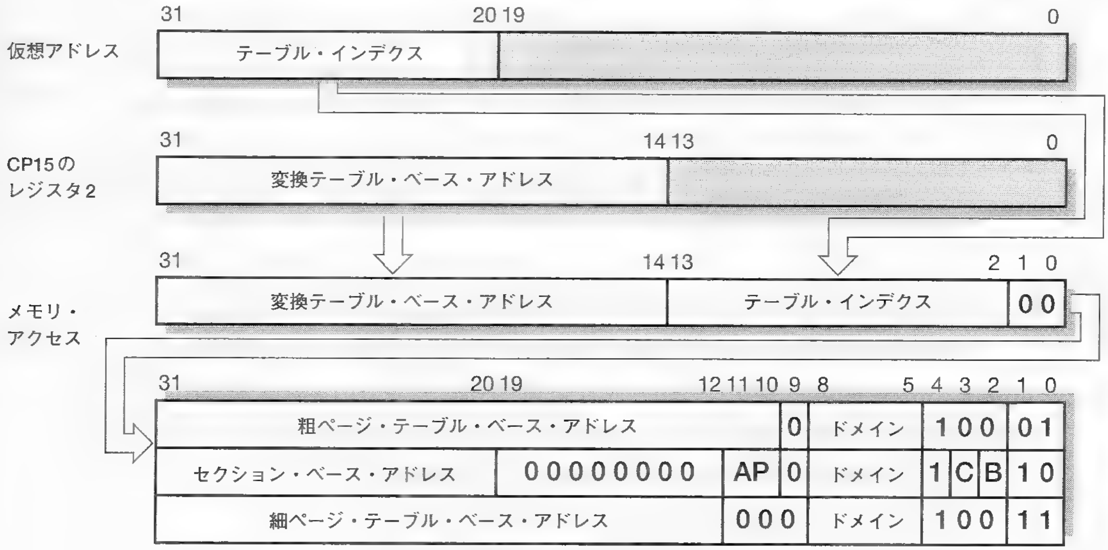
新しい仮想アドレスの変換は，つねに第1レベル・フェッチで始まります(当面TLBは無視する．TLBは下記のプロセスを高速化するだけの一種のキャッシュだからである)．これには，`CP15`のレジスタ2にある変換

〔表11.4〕
ドメイン・アクセス制御ビット

値	ステータス	記述
00	ノー・アクセス	アクセスによってドメイン・フォールトが発生する
01	クライアント	ページとセクション許可ビットがチェックされる
10	予約	未使用
11	マネージャ	ページとセクション許可ビットがチェックされない

ベース・レジスタを使用します．変換ベース・レジスタの[31:14]ビットが仮想アドレスの[31:20]ビットと連結し，図11.3のように第1レベル記述子にアクセスするメモリ・アドレスを形成します．

第1レベル記述子は，セクション記述子または第2レベル・ページ・テーブルへのポインタで，最下位2ビットによって決まります．“01”は第2レベルの粗ページ・テーブルへのポインタを示し，“10”はセクション記述子を示します．“11”は第2レベルの細ページ・テーブルへのポインタを示します(特定のCPUしかサポートしない)．“00”は変換フォールトを引き起こす記述子を示します．



〔図11.3〕第1レベル変換フェッチ

◆セクション変換

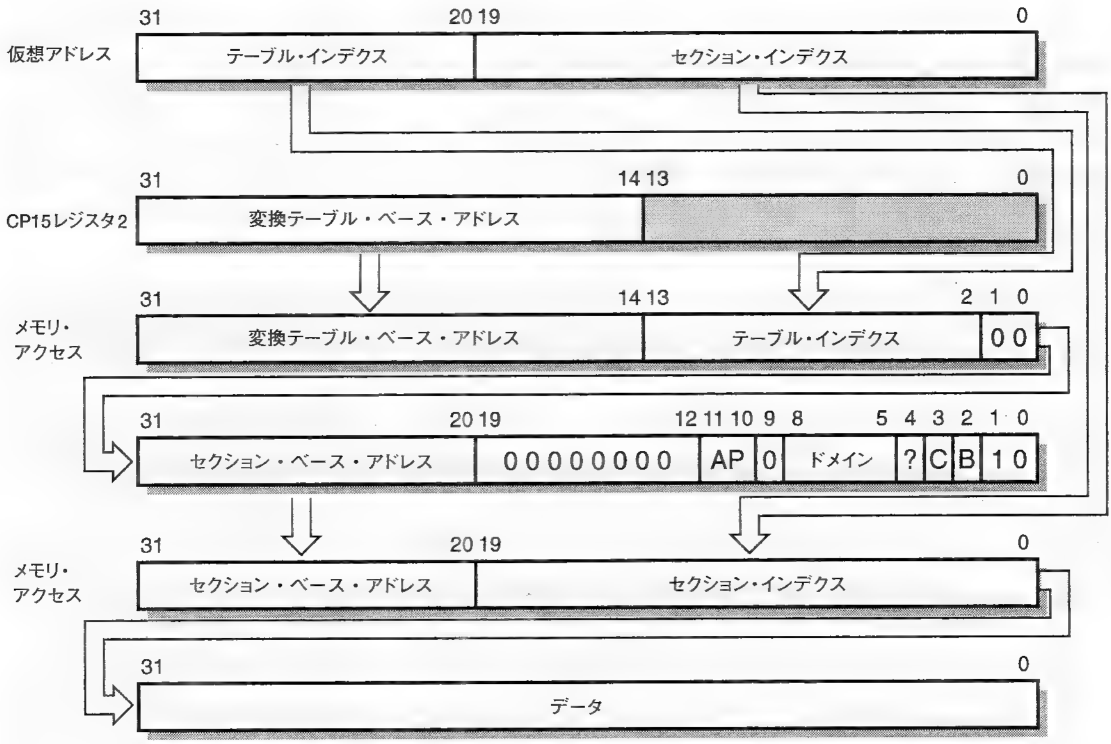
仮想アドレスをセクションに変換することを第1レベル記述子が示している場合，ドメイン(セクション記述子のドメイン)がチェックされ，現在のプロセスがドメインのクライアントである場合はアクセス許可(セクション記述子のAP)もチェックされます．アクセスを許可すると，セクション記述子の[31:20]ビットと仮想アドレスの[19:0]ビットを連結してメモリ・アドレスを作ります．このアドレスはメモリ内のデータにアクセスするのに使用します．完全なセクション変換シーケンスは図11.4のとおりです．

アクセス許可ビット(AP)の動作は，次項「アクセス許可」で説明します．また，バッファ可能ビット(B)，キャッシュ可能ビット(C)は，本章の後の項「キャッシュおよびライト・バッファ制御」で説明します．

◆ページ変換

仮想アドレスをページに変換することを第1レベル記述子が示している場合，第2レベルの粗ページ・テーブルへのアクセスがさらに必要です．第2レベル・ページ記述子のアドレスは，第1レベル記述子の[31:10]ビットと仮想アドレスの[19:12]ビットを連結して作ります．第2レベルの細ページ記述子は，第1レベル記述子の[31:12]ビットと仮想アドレスの[19:10]ビットを連結して作ります．

第2レベルの粗ページ記述子は，ラージ・ページ(64Kバイト)記述子またはスモール・ページ(4Kバイト)記述子で，最下位2ビットによって決まります．“01”はラージ・ページを示し，“10”はスモール・ページを示し



〔図11.4〕セクション変換シーケンス

ます．他の値はトラップされ，"00"は変換フォールトを生成するのに使用します．"11"は使用されません．第2レベルの細ページ記述子は，タイニー・ページ(1Kバイト)記述子であることもあり，この場合，最下位2ビットが"11"になります．上記のようにラージ・ページまたはスモール・ページの記述子であることもあります．

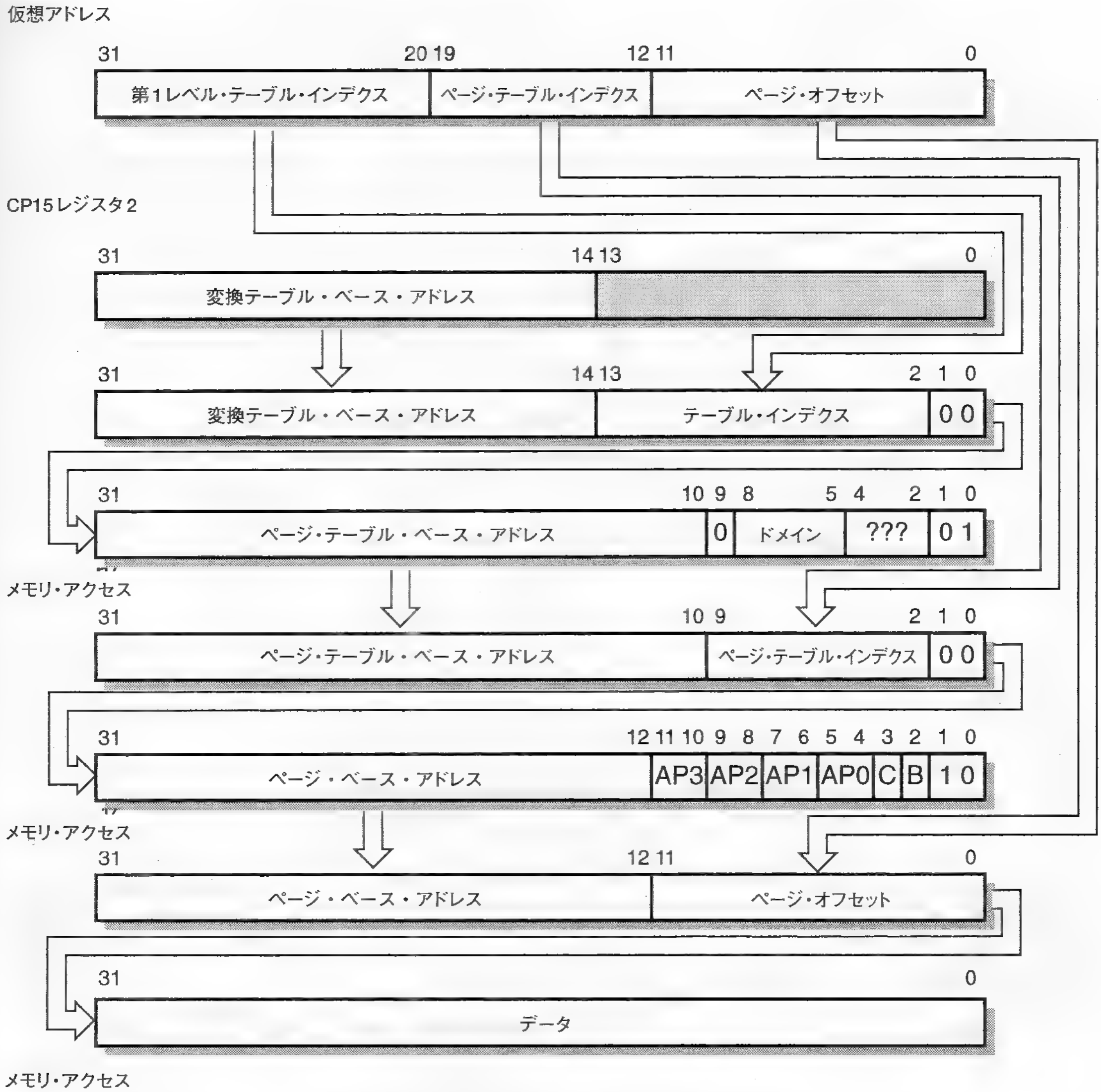
スモール・ページ・ベース・アドレスは，ページ記述子の[31:12]ビットに保持されます．[11:4]ビットは，サブページがページの4分の1サイズの場合，四つの各サブページにつき二つのアクセス許可ビット(AP0-3)をもちます．[3:2]ビットは「バッファ可能」および「キャッシュ可能」ビットをもちます('?'マークのついたビットは，実装により用途が決まる)．

スモール・ページの変換シーケンス全体を図11.5に示します．ラージ・ページの変換シーケンスも類似していますが，仮想アドレスの[15:12]ビットがページ・テーブル・インデクスとページ・オフセットの両方で使用されている点が異なります．したがってラージ・ページの各ページ・テーブル・エントリは，ページ・テーブル・インデクス内にあるこれらのビットそれぞれの値について，ページ・テーブルに16回コピーされることになります．

タイニー・ページ変換方式も同様ですが，細かい第1レベル記述子から開始する必要があります．タイニー・ページはサブページをサポートしていないため，第2レベル記述子には1セットのアクセス許可しかありません．

◆アクセス許可

各セクションまたはサブページのAPビットは，第1レベル記述子のドメイン情報，CP15のレジスタ3のドメイン制御情報，CP15のレジスタ1のSおよびR制御ビット，プロセッサのユーザ・スーパバイザ状態とと



〔図11.5〕スモール・ページ変換シーケンス

もに用いて，アドレス指定された位置への読み出しまたは書き込みアクセスを許可するかどうかを決定します．許可チェック動作は次のように進行します．

(1) アラインメント・チェックがイネーブルであれば(CP15レジスタ1のビット1がセットされている)，アドレス・アラインメントをチェック，間違ってアラインメントされていれば(つまり，ワードが4バイト境界で，ハーフ・ワードが2バイト境界でアラインされていない場合)フォールトを起こします．

(2) 第1レベル記述子の[8:5]ビットからアドレシング指定された位置のドメインを識別します(記述子が無効の場合，第1レベル記述子のフェッチは失敗する)．

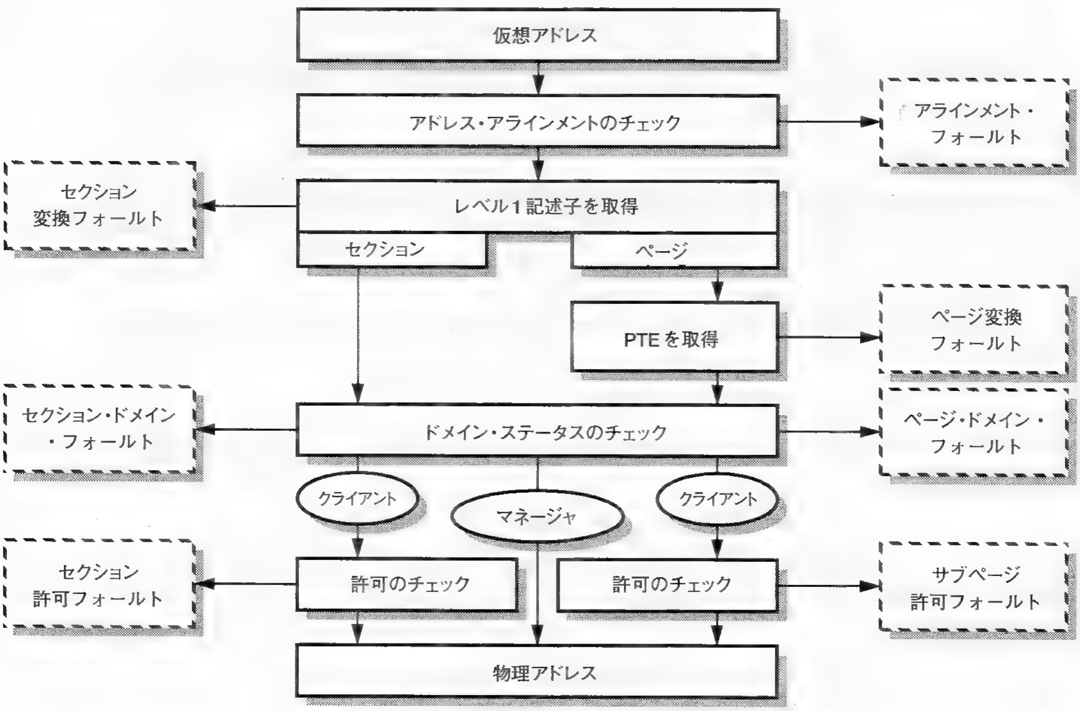
(3) ドメイン・アクセス制御レジスタであるCP15のレジスタ3をチェックし，現在のプロセスがこのドメインのクライアント，マネージャのどちらかを調べます．どちらでもなければ，フォールトを起こします．

(4) もし，このドメインのマネージャであれば，アクセス許可を無視して進みます．クライアントであれば，CP15 レジスタ1のSおよびRビットを用いて，表11.5にしたがってアクセス許可をチェックします．アクセスが許可されないときはフォールトを示し，許可されればデータ・アクセスを続行します．

許可チェック方式は図11.6のとおりです．ここには，アドレス変換過程で生成されうる多くのフォールトが示されています．MMUは，アラインメント，変換，ドメイン，許可に対するフォールトを生成します．さらに，外部メモリ・システムは，キャッシュ・ライン・フェッチ(全CPUがこれをサポートしているわけではない)，キャッシュなしまたはバッファなしアクセス(バッファ書き込みのアボートはサポートしない)，変換テーブル・アクセスでもフォールトを起こすことがあります．これらのフォールトはすべて「アボート」と呼ば

〔表11.5〕アクセス許可

AP	S	R	スーパバイザ	ユーザ
00	0	0	ノー・アクセス	ノー・アクセス
00	1	0	読み出しのみ	ノー・アクセス
00	0	1	読み出しのみ	読み出しのみ
00	1	1	未使用	
01	—	—	読み出し／書き込み	ノー・アクセス
10	—	—	読み出し／書き込み	読み出しのみ
11	—	—	読み出し／書き込み	読み出し／書き込み



〔図11.6〕アクセス許可チェック方式

れ，アクセスが命令，データのどちらに対してかによって，プリフェッチ・アボート例外またはデータ・アボート例外としてプロセッサによって処理されます．

データ・アクセスにおけるフォールトは，フォールト・ステータス・レジスタ（CP15のレジスタ5）とフォールト・アドレス・レジスタ（CP15のレジスタ6）を更新し，フォールトの原因と位置に関する情報を提供します．命令アクセスのフォールトは，命令が実行されるときに（命令が実行されれば）例外を起こすだけで（分岐直後にフェッチされると，実行されないこともある），フォールト・ステータスおよびフォールト・アドレス・レジスタを更新しません．フォールト・アドレスはリンク・レジスタの復帰アドレスから推測します．

◆キャッシュおよびライト・バッファ制御

第2レベル・ページ記述子とセクションのCおよびBビットは，セクションまたはページのデータがキャッシュにコピーされ，さらに（もしくは）ライト・バッファを通してメモリにライトバックされるかどうかを制御します．

キャッシュがライトスルー方式を使う場合は，データがキャッシュ可能かどうかをC，書き込みがバッファできるかどうかをBが制御します．キャッシュがコピーバック方式を使う場合は，代わりに「キャッシュ，バッファなし」の組み合わせを使用し，「ライトスルー・バッファ」動作を指定します（キャッシュ用語については，10.2節の「書き込み方法」の項を参照）．

◆外部フォールト

プロセッサは，バッファ書き込み時に示される外部フォールトから回復できないことに気をつけてください．なぜなら，フォールトが発される前にプロセッサがいくつかの命令を実行しているため，状態回復して失敗したストア命令を再試行できないからです．回復が必要な場合（プロセッサがバス・フォールトに続くストア命令を再試行するときなど）は，バッファしない書き込みを使います．

多くのARMアプリケーションでは，回復可能な外部フォールト要因はありませんから，問題にはなりません．

11.7 同期

データ構造を共有する複数のプロセスを実行するシステムでは，共有データへのアクセスを制御して正しい動作が行われるようにするのが，一般的な問題です．

たとえばあるシステム内で，一つのプロセスがセンサの値群をサンプリングしてメモリにストアしており，別のプロセスが好きなときにそれを使用するとします．後者のプロセスがつねに単一のスナップ・ショットとして値を見ることが重要な場合，値が部分的に変更されたとき前者のプロセスがスワップアウトされたり後者のプロセスがスワップインされたりしないよう注意しなければなりません．これを可能にするメカニズムが**プロセス同期**です．必要なのはデータ構造への「相互排他」です．

◆相互排他

あるプロセスが共有データ構造に処理を実行しようとしていて，その処理が共有データ構造に他プロセスからのアクセスを許したくない場合，まず他プロセスがデータにアクセスしていないときまで待ち，そして処理終了まで他プロセスがアクセスしてこないように何らかのロックをかけることが必要です．

相互排他を可能にする方法の一つは，特定のメモリ位置を用いてデータ構造へのアクセスを制御することで

す．たとえば，その位置のブール値（フラグ）でデータ構造が現在使用中かどうかを示すことができます．そのデータ構造を使用したいプロセスは，それが「フリー」になるまで待ち，使用している間は「ビジー」にマークにします．そして使用が終了すれば，再び「フリー」にマークにします．問題は，構造がフリーになってから「ビジー」にマークされる間に割り込みが起こることです．割り込みはプロセス・スイッチを引き起こし，新しいプロセスはフリーな構造に「ビジー」マークを付け，構造に変更を加えます．そこで，別の割り込みが発生して最初のプロセスに制御を戻すと，そのプロセスは構造がフリーだと誤って信じていることになります．

この問題を解決する標準的な方法は，ブール値がテストおよびセットされている間，割り込みをディセーブルすることです．これはうまくいきますが，保護されたスーパバイザ・モードをもつプロセッサ（ARMなど）では，ユーザが割り込みをディセーブルできません．システム呼び出しが必要になり，完了してユーザプロセスに制御を戻すまでに数クロック・サイクルかかります．

◆SWAP

もっと効率的な方法は**不可分**（つまり割り込みによって分割されない）な「テスト・セット」命令を使うことです．ARMの`SWAP`命令（5.13節を参照）は，まさにこの目的で命令セットに入っている命令です．レジスタを「ビジー」値にセットし，その後ブール値を含むメモリ位置とスワップするのです．ロードした値が「フリー」ならプロセスは続行します．「ビジー」ならプロセスは待ちます．その際，ロックのところを**スピン**（spinning；「フリー」結果が出るまでテストを繰り返す）することもよくあります．

`SWAP`をARM命令セットに入れているのは，ただこの目的のためです．プロセッサの性能にも影響しませんし，動的使用頻度はわずかです．まさにこの機能性だけのために存在しているのです．

11.8　コンテクスト・スイッチ

プロセスはあるコンテクストで動作します．これは，プロセスが正しく動作するために確立していなければならないシステム状態すべてです．状態には，次のものが含まれます．

(1) プログラム・カウンタ，スタックポインタなど，プロセッサ・レジスタすべての値．
(2) そのプロセスが使っている場合は，浮動小数点レジスタの値．
(3) メモリ内の変換テーブル（ただし，TLBの内容はたんなるメモリ内の値のキャッシュであり，使用につれて自動的にアクティブな値を再ロードするため除く）．
(4) メモリ内でプロセスが使用するデータ値（キャッシュの値は，必要に応じて自動的に再ロードされるため除く）．

プロセスがスイッチするときは，旧いプロセスのコンテクストを保存し，新しいプロセスのものを復元します（初めてスタートするのではなく，再開する場合）．

◆いつスイッチするか

コンテクスト・スイッチは外部割り込みの結果により起こることがあります．次に例をあげます．

(1) タイマ割り込みにより，OSがタイムスライス・アルゴリズムにしたがって新しいプロセスをアクティブにするとき．
(2) 特定の事象を待っている優先順位の高いプロセスがその事象に反応して再びアクティブになるとき．

逆に，仕事がなくなったプロセスがOSを呼び出し，外部事象が起こるまで休眠することもあります．

いかなる場合でも，OSが制御をとったり与えたりし，旧いコンテクストの保存，新しいコンテクストの復

元を実行します．ARMベースのシステムでは通常，これはプロセッサがスーパバイザ・モードにあるとき起こります．

◆レジスタ状態

すべてのコンテクスト・スイッチが，*IRQ*，内部フォールト，スーパバイザ・コールへの反応として起こり，スーパバイザ・コードが割り込みを再イネーブルしないならば，プロセス・レジスタ状態はユーザ・モード・レジスタに限られることになります．コンテクスト・スイッチが*FIQ*に反応して起こるか，スーパバイザ・コードが割り込みを再イネーブルするとすると，特権モード・レジスタも保存，リストアする必要が出てきます．

ARMにおけるレジスタ保存および復元の「アーキテクチャ・サポート」は，特権モードからのユーザ・レジスタを保存，復元することの困難さを認識しており，このタスクを助ける特別な命令をもっています．これらの命令は複数ロードおよび複数ストア命令（5.12節を参照）の特別形式で，非ユーザ・モードでコードを動作させ，非ユーザ・モード・レジスタがアドレス指定するメモリ領域からユーザ・レジスタを保存，復元するのを可能にします．

これらの命令がなければ，OSはバンク・ユーザ・レジスタを保存または復元するためにユーザ・モードにスイッチし，その後保護バリアを通ってスーパバイザ・モードに返ってこなければならなくなります．この方法も可能ではあるのですが，非効率的です．

◆浮動小数点状態

ハードウェア・コプロセッサに保持されているにしても，ソフトウェア・エミュレータによってメモリに保持されているにしても，浮動小数点レジスタはそれを使用するプロセス状態の一部を表します．プロセス・スワップごとにそれらの保存，復元のコンテクスト・スイッチ・オーバヘッドがかかるのを防ぐため，OSは，浮動小数点を使うプロセスがスワップアウトされると浮動小数点システムのユーザ・レベル使用をディセーブルします．新しいプロセスが浮動小数点システムを使おうとすると，最初の使用がトラップされます．その時点で，OSは旧いプロセス状態を保存し，新しいものを復元します．そして，浮動小数点システムを再イネーブルして新しいプロセスが自由に使用できるようにします．こうすると，浮動小数点コンテクスト・スイッチ・オーバヘッドは，本当に必要な場合にだけ起こることになります．

◆変換状態

新旧プロセスが独立した変換テーブルをもつ場合，「重量級」のプロセス・スイッチが必要になります．変換テーブル構造全体をスイッチするには，`CP15`のレジスタ2にある第1レベル・ページ・テーブル・ベース・アドレスを変更するだけですみますが，こうすると既存のTLBと（仮想アドレスで指定された）キャッシュ・エントリは無効になるので，これらをフラッシュしなければなりません．TLBと命令またはライトスルー・データキャッシュは全エントリを無効にマークするだけで簡単にフラッシュでき，ARMプロセッサ・チップでは各TLBまたはキャッシュについて`CP15`命令一つが必要なだけですが，コピーバック・キャッシュではすべてのダーティ・ラインを一掃しなければならず，多くの命令が必要になります（物理アドレス・キャッシュの場合この問題は起こらないが，現在のところ全ARM CPUは仮想アドレス・キャッシュを使用している）．

新旧プロセスが同じ変換テーブルを共有する場合，「軽量級」のプロセス・スイッチが必要です．ARMのMMUアーキテクチャの「ドメイン」メカニズムにより，`CP15`のレジスタ3を一度更新するだけで，仮想アドレス空間にある16の異なるサブセットの保護状態が再設定できます．

キャッシュが保護システムの穴にならないために，キャッシュ・アクセスは，許可チェックを伴わなければ

なりません．これは，ドメインとアクセス許可情報を各キャッシュ・ラインのデータと一緒にストアすることでも可能ですが，現在のARMプロセッサはキャッシュ・アクセスと並列にMMUにある情報を使って許可をチェックしています．

11.9 入出力

入出力(I/O)機能は，メモリ・マップされたアドレス可能周辺レジスタと割り込み入力の組み合わせで，ARMシステム上に実装されています．ダイレクト・メモリ・アクセス(DMA)ハードウェアを含むARMシステムもあります．

◆メモリ・マップされた周辺装置

シリアル・コントローラなどの周辺デバイスは，多くのレジスタをもっています．メモリ・マップされたシステムでは，これらの各レジスタが特定のアドレスにおけるメモリ位置のように見えます(他のシステム構成では，I/O機能がメモリ・デバイスと別のアドレス空間にあることもある)．

たとえばシリアル・コントローラは，次のようなレジスタ・セットをもつことがあります．

(1) 送信データ・レジスタ(書き込み専用)

ここに書かれたデータがシリアル・ラインで送られます．

(2) 受信データ・レジスタ(読み出し専用)

シリアル線で到着するデータがここに現れます．

(3) 制御レジスタ(読み出し/書き込み用)

データ速度を設定し，RTS(送信要求)などの信号を管理します．

(4) 割り込みイネーブル・レジスタ(読み出し/書き込み用)

どのハードウェア事象が割り込みを生成するかを制御します．

(5) ステータス・レジスタ(読み出し専用)

読み出しデータが使用可能かどうか，ライト・バッファがフルかどうかなどを示します．

データを受信するには，ソフトウェアがデバイスを適切にセットアップし，データを入手したりエラー状態を検知したりすると割り込みを生成するようにしなければなりません．そして，割り込みルーチンがデータをバッファにコピーし，エラー状態をチェックします．

◆メモリ・マップ問題

メモリ・マップされた周辺レジスタの動作はメモリと異なることに注意してください．読み出しデータ・レジスタの2連続読み出しは，その位置に書き込みがなくても異なる結果になります．本当のメモリの読み出しは再現性がありますが，周辺装置への読み出しは現在値をクリアし，次の値は異なることがあります．そのような位置は**読み出し依存**(read-sensitive)といいます．

読み出し依存の位置を含むプログラムは注意して書く必要があります．とくに，そのような位置は決してキャッシュ・メモリにコピーしてはなりません．

多くのARMシステムでは，I/O位置はユーザ・コードに対してアクセス不可にするので，デバイスにアクセスする唯一の方法はスーパバイザ・コール(SWI)またはこれらの呼び出しを使用するために書かれたCライブラリ関数だけです．

◆ダイレクト・メモリ・アクセス（DMA）

I/O機能が広いデータ帯域幅をもつとき，プロセッサ性能のかなりの部分がI/Oシステムからの割り込み処理で消費されます．多くのシステムは，DMAハードウェアでプロセッサの助けなしに最低レベルのI/Oデータ転送を処理します．一般的に，DMAハードウェアは周辺装置からメモリのバッファ領域へのデータ・ブロック転送を処理し，エラーが起こったりバッファがフルになったりした時だけプロセッサに割り込みます．そのため，プロセッサは，バイトごとに1回ではなく，バッファごとに1回割り込みの面倒を見るだけです．

しかし，DMAデータ・トラフィックはメモリ・バス帯域幅の一部を占有するため，プロセッサ性能はI/O動作によってやはり低減されることに注意してください（もっとも，割り込みでデータ・トラフィックを処理するのよりはかなり少なくなる）．

◆高速割り込み要求

ARMの高速割り込み（*FIQ*）アーキテクチャには，他の例外モードより多くのバンク・レジスタがあり（図2.[illegible]参照），例外処理にかかるレジスタ保存，復元オーバヘッドを最小化しています．レジスタ数は，DMAチャネルのソフトウェア・エミュレーションの実装に必要な数になるよう決定されています．

DMAサポートのないARMシステムで，かなり高いバンド幅要求をもつI/Oデータ・トラフィックのソースが一つあるとすると，*FIQ*割り込みをこのソースに割り当て，*IRQ*で他の全ソースをサポートすることを考慮したほうがいいでしょう．複数の異なるデータ・ソースに同時に*FIQ*を使うことは非常に非効率的ですが，粗い粒度でソース間をスイッチするのは適切です．

◆割り込み遅延時間

プロセッサの重要なパラメータの一つに**割り込み遅延時間**（interrupt latency）があります．これは，割り込み応答に最悪の場合でどのくらい時間がかかるかを示します．ARM6では，最悪の場合の*FIQ*待ち時間は次の要素によって決まります．

1）**要求信号がFIQ同期ラッチを通過する時間**：3クロック・サイクル（最悪の場合）
2）**最長の命令を完了する時間**（16レジスタの複数ロード）：20クロック・サイクル
3）**データ・アボート・エントリ・シーケンスにかかる時間**：3クロック・サイクル（データ・アボートは*FIQ*より優先順位が高いが，*FIQ*をマスクアウトしないことに注意．5.2節の「例外優先順位」の項を参照）
4）**FIQエントリ・シーケンスにかかる時間**：2クロック・サイクル

したがって，最悪の場合の総遅延時間は28クロック・サイクルです．この後，ARM6は0x1C，つまり*FIQ*エントリ・ポイントで命令実行を開始します．これらのサイクルはシーケンシャルのこともあれば非シーケンシャルのこともあり，低速メモリ・デバイスをアドレスする場合はメモリ・アクセスがさらに遅れることもあります．最良の場合の遅延時間は，4クロック・サイクルです．

*IRQ*遅延時間の計算も同様ですが，最長の*FIQ*ルーチンを完了するのにかかる遅延分を含めなければなりません（*FIQ*は*IRQ*より優先順位が高いため）．

◆キャッシュI/Oインタラクション

普通，キャッシュはプロセッサを高速化するものとされています．適度な期間をとって性能を平均すれば，たいていこれは正しくなります．しかし，多くの場合，割り込みが使用されるのは，最悪の場合のリアルタイム応答がクリティカルであるところです．そのような場合，キャッシュは性能を大幅に低下させます．そのうえ，MMUはもっと状況を悪化させてしまうのです．

ARM710で，最悪の場合の割り込み総遅延時間を見てみましょう．ARM710については次章で触れます．遅延時間は次の総計になります．

(1) 要求信号が*FIQ*同期ラッチを通過する時間：前と同じく3クロック・サイクル（最悪の場合）

(2) 最長の命令を完了するまでの時間（16レジスタの複数ロード）：20クロック・サイクル

しかし，これはライト・バッファをフラッシュするため，さらに12サイクルかかります．そしてMMUのTLBミスが3回起こり，また18クロック・サイクル．6キャッシュ・ミスに24サイクル．当初の20サイクルはライン・フェッチとオーバラップしますが，それでもこのステージの総計は66サイクルに上ることもあるのです．

(3) データ・アボート・エントリ・シーケンスにかかる時間：3クロック・サイクル

しかし，ベクタ空間からのフェッチがMMUミスとキャッシュ・ミスを加え，結局12サイクルかかることがあります．

(4) FIQエントリ・シーケンスにかかる時間：2クロック・サイクル

しかし，別のキャッシュミスを引き起こすと6サイクルになります．

総計は今や87クロック・サイクルとなり，その多くは非シーケンシャル・メモリ・アクセスです．メモリ階層で汎用プログラムを平均すると高速化する自動メカニズムは，クリティカル・コード・セグメントで最悪の場合を考えたとき，しばしば逆効果を起こすことに注意してください．

◆遅延時間短縮

リアルタイム制約をたんに満足させるだけなら，遅延時間はどのように短縮すればよいでしょうか？

(1) 高速オンチップRAMの固定領域（メモリ下位でベクタ空間を含むものなど）は，例外エントリを高速化する．

(2) TLBとキャッシュ・セクションは，クリティカル・コード・セグメントがミス・ペナルティを負わないように**ロック・ダウン**することができる．

キャッシュとMMUが有効な汎用システムでさえ，しばしばクリティカルなリアルタイム制約があることに注意してください．たとえば，ディスク・データ・トラフィックの処理やLANの管理などです．とくに，DMAハードウェア・サポートが少ない低価格システムに当てはまります．

◆キャッシュに関する他の問題

キャッシュが存在するとき，気をつけなければならないことは他にもあります．以下，例をあげます．

(1) キャッシュは，アドレスに新しい値が書き込まれるまでは何度読んでも同じ値が返ってくることを前提としている．I/Oデバイスは，そうではない．読むたびに異なるデータを返してくる．

(2) キャッシュは，1回につき1データ・ブロック（普通は4ワード）をシーケンシャル・アドレスからフェッチする．I/Oデバイスは，連続するアドレスでしばしば異なるレジスタ機能をもつので，全部一度に読むと予測していない結果になる．

そのため，メモリのI/O領域は通常キャッシュ不可とマークされ，アクセスはキャッシュをバイパスするのです．一般的に，キャッシュは読み出し依存のデバイスではうまく作用しません．ディスプレイ・フレーム・バッファも注意が必要で，しばしばキャッシュ不可になっています．

◆OSの問題

普通，I/Oデバイス・レジスタの低レベルな詳細や割り込み処理はOSの仕事です．多くのプロセスは，次のバイトを`r0`にロード，そのあと適切なスーパバイザ・コールをすることによってデータをシリアル・ポー

線に送ります．OSはデバイス・ドライバと呼ばれるサブルーチンを呼び出し，伝送バッファが空かどうか，線がアクティブか，伝送エラーが起こっていないか，などをチェックします．プロセスがポインタをOSに渡し，大量の完全なバッファを出力させる呼び出しさえあります．

データが詰まったバッファをシリアル線で送るには多少時間がかかるため，OSは，伝送バッファに次のデータを入れる空間ができるまで，プロセスに制御を戻すことがあります．シリアル・ハードウェア・デバイスからの割り込みがOSに制御を戻し，伝送バッファを再び満たしてから，割り込まれたプロセスに制御を戻します．さらに割り込み，転送と，バッファ全部が送られるまで続きます．

シリアル線動作を要求したプロセスに仕事がなくなったり，タイマや他ソースからの割り込みが他プロセスをアクティブにすることもあります．OSは，データ・バッファを自分でアクセス不可にならないよう，変換テーブルに変更を加えるときに注意しなければなりません．また，2番目のプロセスからの要求でデータをシリアル線に送るときは注意し，最初のプロセスからの転送が進行しているのを妨害しないようにしなければなりません．共有資源の使用において競合がないようにするには**資源割り当て**(resource allocation)が用いられます．

プロセスが出力機能を要求してから出力が完了するまで，非アクティブになったり，特定の入力が到着するまで非アクティブになったりすることがあります．要求をOSに預け，入出力事象が起こったときに再びアクティブにすることも可能です．

11.10　例題と演習

<例題11.1>　ARMでは，なぜユーザ・レベル・コードが割り込みをディセーブルできないのか？

ユーザが割り込みをディセーブルできるようにすると，保護されたOSの構築が不可能になります．次のコードは，悪意のあるユーザが現在アクティブなプログラムすべてを破壊する方法を示しています．

```
        MSR     CPSR_f, #&c0      ; disable IRQ and FIQ
HERE    B       HERE              ; loop forever
```

いったん割り込みがディセーブルされると，OSが制御を取り戻す方法はありませんから，プログラムは永遠にループし続けます．止める方法は現在アクティブな全プログラムを破壊するハード・リセットしかないのです．

ユーザが割り込みをディセーブルできない場合，OSはタイマからの定期的割り込みを設定できるため，無限ループは割り込まれ，OSは他のプログラムをスケジュールできます．このプログラムでは，OSが費やせるCPU時間に上限がある場合はタイムアウトになり，さもなければスイッチ・インされるたびにループを続け，課金付きのシステムでは膨大な費用になります．

演習11.1.1　安全なOSで，最下位メモリ(例外ベクタが位置する)には，最低どのくらいの保護レベルが必要ですか？

演習11.1.2　ARMにSWAP命令がないとして，同期をサポートするハードウェア周辺装置を考案しなさい(ヒント：標準メモリは役に立たない．つまり，その位置が読み出し依存でなければならない)．

第12章 ARM CPUコア

ARMアプリケーションの中には，単純な整数プロセッサ・コアを基本処理コンポーネントに使用しているものもあれば，キャッシュ・メモリやメモリ管理ハードウェアなどの密結合機能を必要とするものもあります．ARM Limitedは，整数コアをベースとしてそのような多様な「CPU」構成を提供しています．

ここで説明するARM CPUコアは，ARM710T，720T，740T，ARM810（現在はARM9シリーズに取って代わられている），StrongARM，ARM920T，940T，ARM 1020Eです．これらのCPUは，さまざまなパイプラインやキャッシュ構成をもち，低消費電力アプリケーション向けプロセッサを設計するうえで起こってくる問題をわかりやすく例示しています．

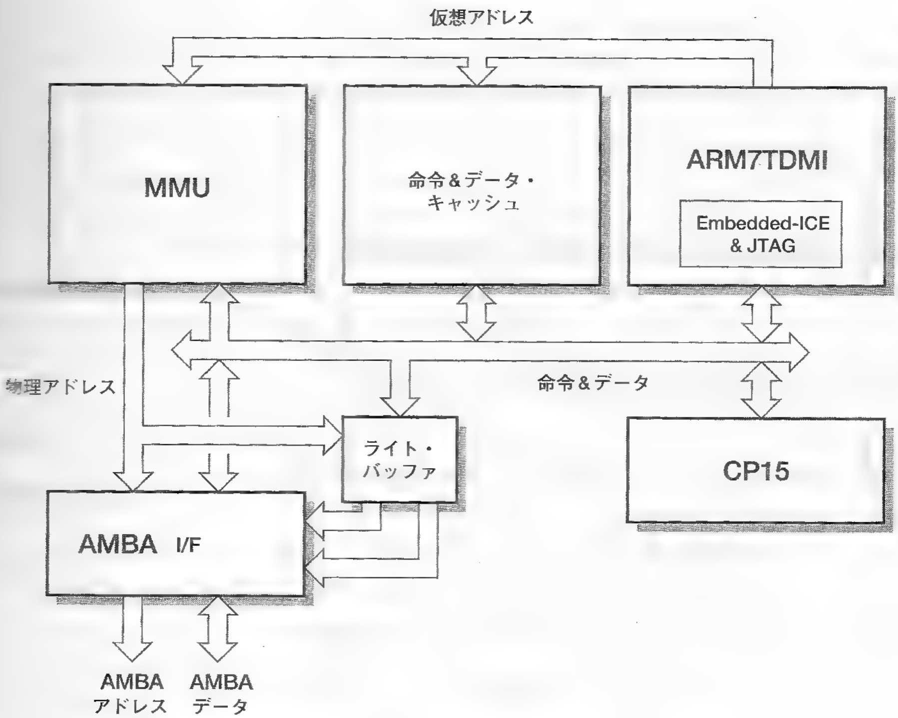
キャッシュ・メモリの主な役割は，プロセッサ・コアの命令およびデータ帯域幅の要求を満たすことですから，キャッシュ構成は使用する特定のプロセッサ・コアに密接に結合しています．システム・オンチップ設計の上では，キャッシュはCPUコアに必要な外部メモリ帯域幅を減らして，オンチップ・バスで処理できる程度にすることが目的です．高性能ARMプロセッサ・コアでも，もしAMBAバスに直接接続されていれば，ARM7TDMIとほとんど変わらない速さになってしまうでしょう．したがって，つねに高速ローカル・メモリやキャッシュとともに使用されるのです．

メモリ管理も，完全変換ベースのシステムであれ，単純な保護ユニットであれ，プロセッサ・コアと密接に結合していなければならない複雑なシステム機能です．ARM CPUコアでは，一つのマクロセルにプロセッサ・コア，キャッシュ，MMU，それに（通常は）AMBAインターフェースが集積されています．

12.1 ARM710T，720T，740T

ARM710T，720T，740Tは，ARM7TDMIプロセッサ・コアをベースとし（9.1節参照），それに8Kバイトの命令およびデータの混在するキャッシュが追加されています．外部メモリとペリフェラルにはAMBAバス・マスタ・ユニット経由でアクセスし，ライト・バッファとメモリ管理（ARM710T，720T）ユニットまたはメモリ保護（ARM740T）ユニットも組み込まれています．

ARM710Tと720TのCPU構成は似ており，図12.1に示すとおりです．



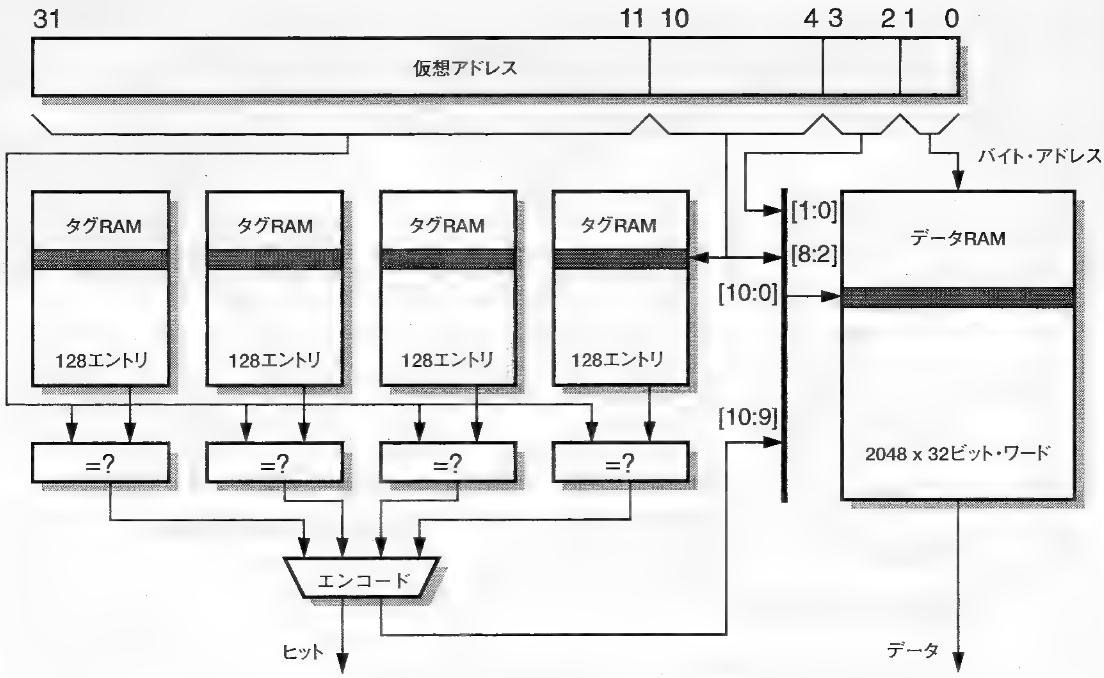
〔図12.1〕ARM710TとARM720Tのキャッシュ構成

◆ARM710Tキャッシュ

ARM7TDMIプロセッサ・コアにはメモリ・ポートが一つしかありませんから，命令とデータを統合したキャッシュと組み合わせるのは当然です．ARM710Tは，8Kバイトの命令とデータを統合したキャッシュを組み込んでいます．キャッシュは，16バイト・ラインで構成されており，4ウェイ・セット・アソシアティブです．ランダム置換アルゴリズムが，キャッシュ・ミス時に四つの位置のうちどれを新しいデータで上書きするかを選択します．ターゲット・クロック速度が標準オフチップ・メモリ・デバイスのたった数倍しか速くないため，キャッシュはライトスルー方式を使用します．

ARM710Tのキャッシュ構成は，図12.2のとおりです．四つのタグ・ストアそれぞれのインデクスに仮想アドレス・ビット[10:4]を使用しています．タグは対応データの仮想アドレス・ビット[31:11]をもつため，現在の仮想アドレス・ビット[31:11]と比較します．タグのうち一つがマッチすると，キャッシュがヒットし，同じインデクス（仮想アドレス・ビット[10:4]）を用いてデータRAMから対応ラインと，マッチしているタグを生成したタグ・ストア番号をエンコードしている2ビットにアクセスが可能になります．仮想アドレス・ビット[3:2]はラインからワードを指定し，バイト・アクセスまたはハーフ・ワード・アクセスが必要な場合はビット[1:0]がワードからバイトまたはハーフ・ワードを指定します．

ARM710Tのキャッシュ構成とARM3やARM610（10.4節に詳細）のキャッシュ構成を比較すると興味深いのは，高性能かつ低消費電力なキャッシュ設計において起こってくる問題がよく現れていることです．この種のアプリケーション向けにキャッシュを設計する正しい方法について結論はまだ出ていませんが，設計者はこれらのARMチップにおける例を観察して参考にすることができます．



〔図12.2〕ARM710Tのキャッシュ構成

◆キャッシュ速度

高連想度キャッシュは，ヒット率がもっとも高くなりますが，CAMとその後にRAMのシーケンシャル・アクセスが必要なため，サイクル時間の最高速度が制限されます．低連想度キャッシュにするとタグおよびデータの並列アクセスを実行するため，サイクル時間を速くできます．ダイレクト・マップ・キャッシュはフルアソシアティブ・キャッシュよりかなりヒット率が落ちますが，ダイレクト・マップ・キャッシュから2または4ウェイ・セット・アソシアティブ・キャッシュに移行するだけで，連想性による利点はほとんど得られます．4ウェイを超えると，連想性による利点はあまりないのです．しかし，やはりフルアソシアティブCAM-RAMキャッシュのほうが4ウェイ・セット・アソシアティブRAM-RAMキャッシュよりかなり単純です．

◆キャッシュ電力

CAMは，毎サイクルですべてのエントリを並列比較するので，電力を多めに消費します．連想度を少し低くしてキャッシュをセグメント化し，CAMのサブセクションだけをアクティブにすると，多少複雑になるだけで大幅な消費電力節約になります．

SRAMでおもに電力を消費するのは，アナログ・センス・アンプです．4ウェイ・キャッシュはダイレクト・マップ・キャッシュより，タグ・ストア内で4倍のセンス・アンプをアクティブにしなければなりません．並列のタグとデータのアクセスによって得られる速度を生かそうとすると，データ・ストアでも4倍のセンス・アンプを使用します（一方，RAM-RAMキャッシュでは，対応タグ・ストアにヒットを検知したときだけデータRAMがアクティブになり，シリアルなタグとデータのアクセスを実行して電力を節約する）．データが有効になると，すぐセンス・アンプをオフにするセルフタイム・パワーダウン回路を使用することにより，電力の無駄は最小化できますが，センス・アンプはやはり大きな電力を消費します．

◆シーケンシャル・アクセス

プロセッサが同じキャッシュ・ライン内のメモリ位置にアクセスするとき，最初のアクセス以外はタグ・ルックアップをバイパスすることが可能なはずです．ARMは，次のメモリ・アクセスがいつ現在のアクセスとシーケンシャルになるかを示す信号を発行します．そして，ここから現在のアドレスについてアクセスが同じラインにあることを推測することもできます（同じラインの全アクセスを捕らえられるわけではないが，ほとんどが可能で，実装も非常に単純）．

アクセスが同じラインになる場合，タグ・ルックアップをバイパスすることによってアクセス速度を向上し，電力を節約することができます．シーケンシャル・アクセスは，低速センス・アンプを使用して（アナログ回路ではなく標準的な論理を用いて）大幅に電力を節約できる可能性もあります．ビット・ラインの電圧振幅の増加がセンス・アンプで節約した以上の電力を必要としないことを確かめておく必要があります．

◆電力最適化

キャッシュ設計者は，キャッシュ電力だけでなくシステム電力全体を最小化するのが目的だということを念頭に置かなければなりません．オフチップ・アクセスはオンチップ・アクセスより電力を多く消費するので，まずヒット率を上げる構成を見つけることが重要です．高連想度CAM-RAM構成か，セット・アソシアティブRAM-RAM構成かを選択するには，設計上のあらゆる点を検討する必要があります．また，電力効率のよいセンス・アンプ，CAMヒット検出装置を構築する新しい方法など，回路に関する低レベルの問題も大きく影響してきます．

電力節約，性能向上のためシーケンシャル・アクセスを採用することはつねによい考えです．ARMが出した一般的な動的実行の統計を見ると，75%のアクセスはシーケンシャルです．シーケンシャル・アクセスは基本的に処理が簡単ですから，これは見過ごせない数字です．電力効率が最重要事項である場合，性能を犠牲にして非シーケンシャル・アクセスが2クロック・サイクルとるようにするのも割に合います．なぜなら性能は25%しか落ちないのに比べて，キャッシュ電力は半分から3分の1に減るからです．

低電力を研究する中で興味深い問題は，電力効率に関して最高のキャッシュ構成が，高性能に関しても最高の構成になりうるかどうか，ということです．

◆ARM710TのMMU

ARM710Tのメモリ管理ユニットは，11.5節で述べたシステム制御コプロセッサを使って11.6節で述べたARMメモリ管理アーキテクチャを実装しています．

変換ルックアサイド・バッファ（TLB）は，最近使用した変換の64エントリの連想キャッシュであり，2ステージのテーブル・ルックアップの必要性をアクセスの大部分において省くことによって，変換処理を加速します．

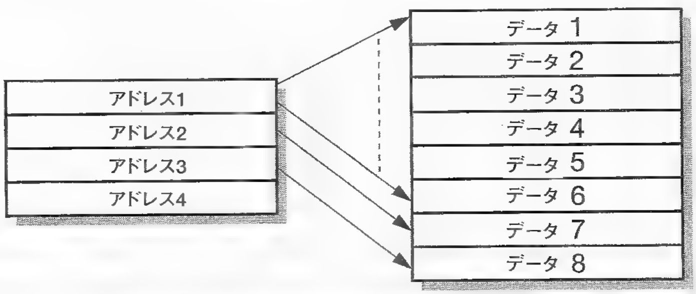
◆ARM710Tのライト・バッファ

ライト・バッファは4アドレスと8データ・ワードを保持します．メモリ管理ユニットは，どのアドレスがバッファ可能かを定義します．各アドレスは任意数のデータ・ワードと結びつくことができますから，ライト・バッファは，たとえば，一つのアドレスに書き込む1ワード（またはバイト）のデータや他のアドレスに書き込む7ワードや別々のアドレスに書き込む4ワード・ブロック二つなどができます．特定のアドレスに対応するデータ・ワードは，そのアドレスで始まるシーケンシャルなメモリ位置に書き込まれます（当然ながら，一つのアドレスに対応する複数のデータ・ワードは，おもに複数レジスタ・ストア命令によって生成される．それ以外に可能性があるのは，外部コプロセッサからのデータ転送だけ）．

マッピングは図12.3のとおりです．ここでは最初のアドレスが6データ・ワードに対応し，2番目と3番目のアドレスが1データ・ワードにそれぞれ対応しています．4番目のアドレスは現在使用されていません．ライト・バッファは全4アドレスが使用されているか，全8データ・ワードがフルの場合にフルとなります．

プロセッサは，高速（キャッシュ）クロック速度でバッファに書き込み，ライト・バッファがメモリ・クロック速度でデータをメモリにストアしている間もキャッシュから命令を実行し続けることができます．したがってプロセッサは，必要な命令とデータがキャッシュにあり，書き込み動作を実行するうえでライト・バッファがフルでないかぎり，メモリ速度とは完全に分離しています．そしてライト・バッファは，ハードウェア・コストがあまりかからず，性能は約15%向上します．

ライト・バッファの主な欠点は，プロセッサ状態が回復可能でないため，バッファされた書き込みによって起こった外部メモリ・フォールトから回復できないことです．それでも変換フォールトはオンチップMMUで検出され，データがライト・バッファに到達する前に例外が発生しますから，プロセッサは仮想メモリをサポートすることができます．しかし，バッファされた書き込みがイネーブルされている場合，ソフトウェアによるエラー・リカバリに基づくメモリ・エラー訂正方式はサポートできません．



〔図12.3〕ライト・バッファのマッピング例

◆ARM710Tシリコン

0.35 μm CMOSプロセス上に実装されたARM710Tの特徴は表12.1にまとめてあります．

〔表12.1〕ARM710Tの特徴

プロセス	0.35 μm	トランジスタ数	N/A	MIPS	53
メタル層	3	コア面積	11.7 mm^2	消費電力	240 mW
V_{dd}	3.3 V	クロック	0–59 MHz	MIPS/W	220

◆ARM720T

ARM720TはARM710Tと非常によく似ていますが，以下の拡張があります．

(1) アドレス空間の最下位32Mバイトにある仮想アドレスは，ProcessIDレジスタ（CP15のレジスタ13）に指定されている32Mバイトのメモリ領域に移動することができます．

(2) 例外ベクタをメモリの最下位から0xffff0000に移動し，上記のメカニズムによって変換されないようにす

ることができます．この機能はCP15のレジスタ1の「V」ビットが制御します．

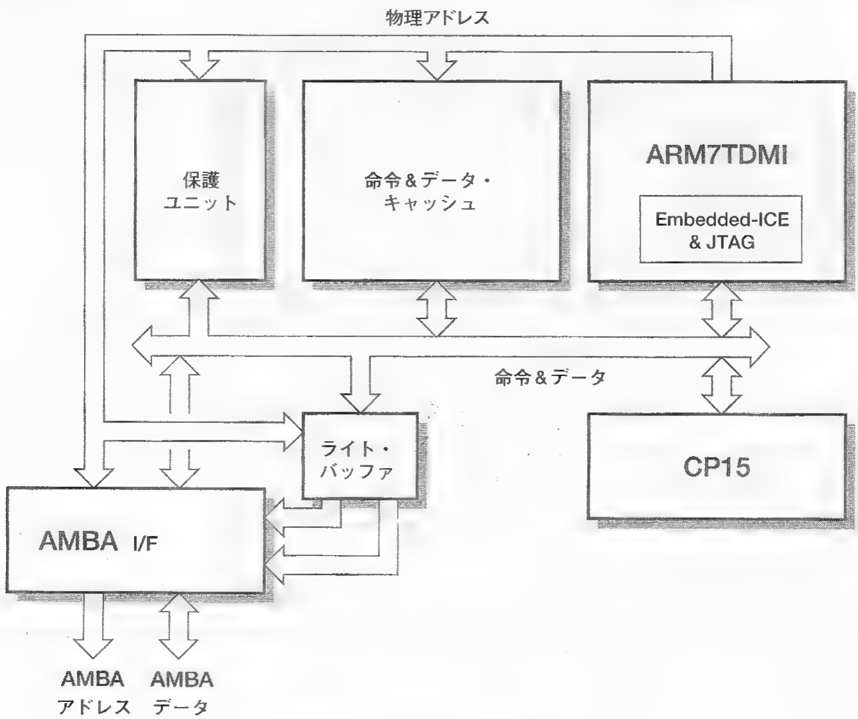
これらの拡張は，CPUコアの能力を向上してWindowsCEオペレーティング・システムをサポートするためのものです．11.5節で述べたCP15 MMU制御レジスタ中に実装されます．

◆ARM740T

ARM740Tは，メモリ管理ユニットの代わりに単純なメモリ保護ユニットをもっている点だけがARM710Tと異なります．メモリ保護ユニットは，11.3節で述べたシステム制御コプロセッサを使って11.4節で説明したアーキテクチャを実装しています．

保護ユニットは，仮想メモリ・アドレスから物理メモリ・アドレスへの変換をサポートしていませんが，基本的な保護とキャッシュ制御機能を低コストで提供します．これは，フルアドレス変換にかかるオーバヘッドを許容できない決まったソフトウェア・システムが動作する組み込み型アプリケーションにとっては有利となります．また，TLBミスは外部メモリ・アクセスをいくつか起こしてしまいますから，アドレス変換ハードウェアを省略することで（必要でない場合に）性能と電力効率も向上します．

ARM740Tの構成は，図12.4のとおりです．



〔図12.4〕ARM740Tの構成

◆ARM740Tシリコン

0.35 μ m CMOSプロセス上に実装されたARM740Tの特徴は表12.2にまとめてあります．

表12.2からわかるように，ARM740Tの一般的な特徴は表12.1にあげたARM710T，720Tのものと同じです．大きな違いは，MMUがないことで2 mm^{2}弱のコア面積を節約していることと，消費電力が少ないことだけです．

〔表12.2〕ARM740Tの特徴

プロセス	0.35 μm	トランジスタ数	N/A	MIPS	53
メタル層	3	コア面積	9.8 mm^2	消費電力	175 mW
V_{dd}	3.3 V	クロック	0–59 MHz	MIPS/W	300

12.2 ARM810

ARM810は，オンチップ・キャッシュとMMUをもつ高性能ARM CPUです．ARM810は，Acorn Computersで設計したARM6，ARM7で使われているパイプライン構造とは根本的に異なるパイプライン構造をもつARM命令セットの最初に実装となったものです．ARM810は現在ARM9シリーズによって取って代わられています．

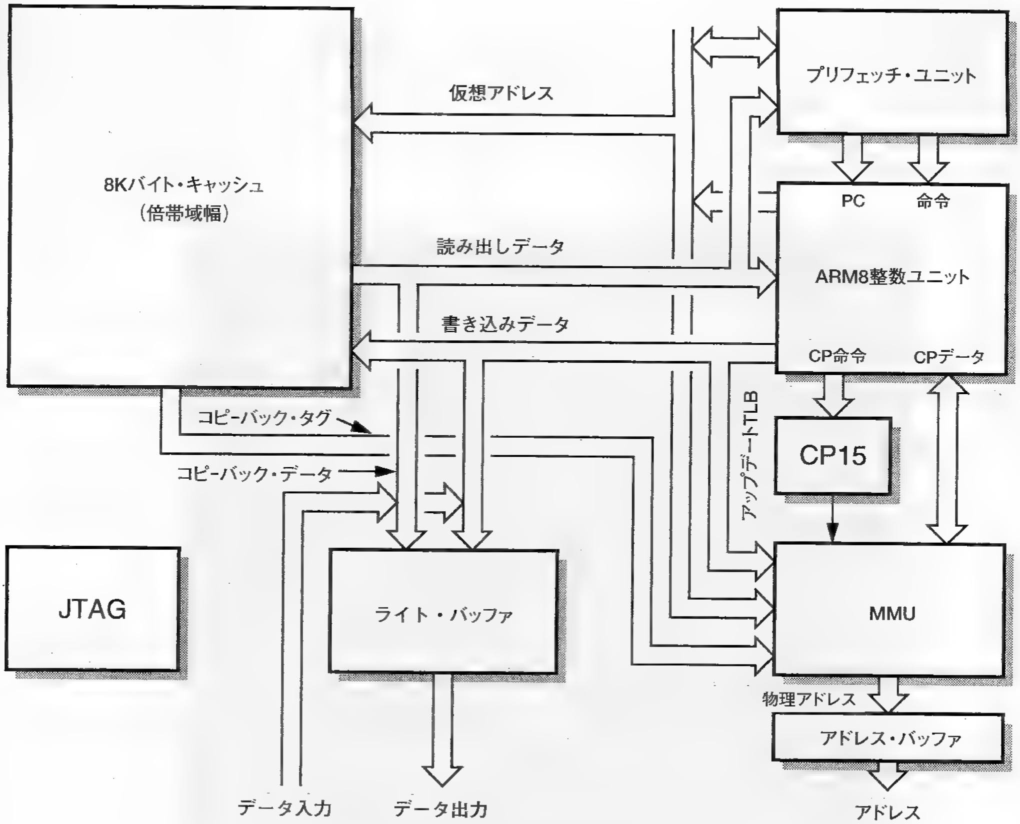
ARM8コアは，ARM810で使用している整数処理ユニットです．これは9.2節で説明しました．ARM810では，基本CPUに次のオンチップ・サポートが加わっています．

(1) 8Kバイトの仮想アドレスで命令とデータが統合されたキャッシュ，コピーバック(もしくはライトスルー．ページ・テーブル・エントリが制御)書き込み方式を使用，ARM8コアに必要な倍帯幅機能をもつ．キャッシュは64ウェイ・アソシアティブで，将来，組み込み型システム・チップ向け小型キャッシュや，より高度なプロセス技術における大型キャッシュの開発を楽にするため，1Kバイト・コンポーネントで構成されている．多くの組み込み型アプリケーションに見られる速度クリティカルなコード・セクションがフラッシュされないよう，そのキャッシュ領域を「ロック・ダウン」できる．

(2) 11.6節で説明したARM MMUアーキテクチャに適合するメモリ管理ユニット．11.5節で説明したシステム制御コプロセッサを使用している．

(3) 外部メモリへの書き込みが完了するまでの間，プロセッサの動作継続を可能にするライト・バッファ．

ARM810の構成は，図12.5のとおりです．



〔図12.5〕ARM810の構成

◆倍帯域幅キャッシュ

コアの必要性に応え，キャッシュは倍帯域幅をもちます．つまり，外部メモリ・アクセスは従来のライン・リフィルと個別データ転送プロトコルを使用します．倍帯域幅はシーケンシャル・メモリ・アクセスの場合のみ使用できるため，プリフェッチ・ユニットは命令フェッチに，コアは複数レジスタ・ロード命令に使用しています．

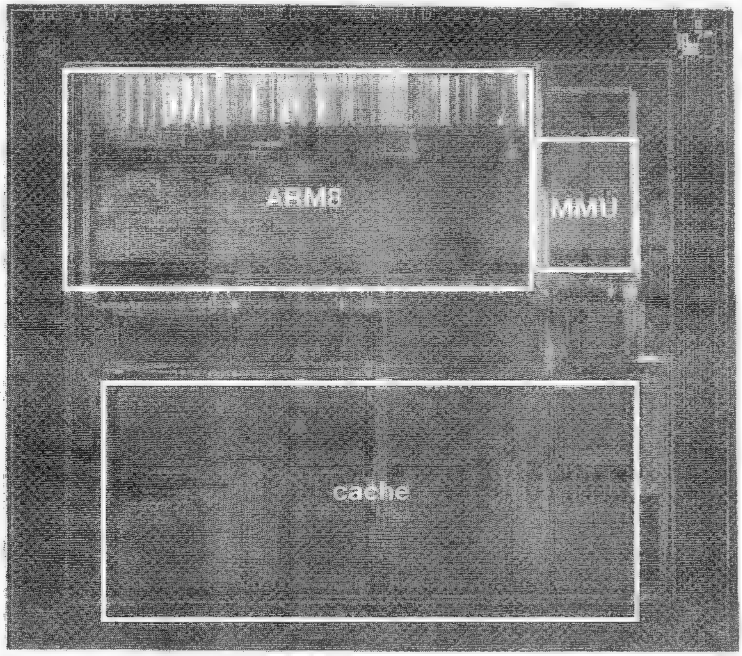
ARM7TDMIのパイプライン構成では毎サイクルでメモリ・インターフェースを使用するため，プロセッサのCPI（命令ごとにかかるクロック数）を向上するには，何とかして使用できるメモリ帯域幅を広げなければなりませんでした．StrongARM（次節で説明）は，別々の命令，データ・キャッシュを組み込み，潜在的に使用可能な帯域幅を倍にしています（ただし，帯域幅全部が使用できるわけではない．なぜならARMが生成する命令トラフィックは帯域幅がデータ・トラフィックの約2倍のため，帯域幅の有効な増加は約50%になるから）．ARM810は，1クロック・サイクルにつき二つのシーケンシャル・ワードを戻すことにより，使用可能な帯域幅を60%も広げました．なぜなら通常，ARMのメモリ・アクセスの約75%がシーケンシャルだからです．ARM810のほうが帯域幅は広くなりましたが，命令とデータ・アクセスが競合する可能性は増しました．これら二つの方法における利点を比較することは，簡単ではありません．

キャッシュはコピーバック書き込み方式を用い，仮想アドレスをもつため，ダーティ・ラインをライトバックするにはアドレス変換が必要です．ARM810のメカニズムでは，仮想タグをMMUに送って変換します．

◆ARM810シリコン

ARM810ダイの写真は，図12.6のとおりです．ARM8コアのデータパスは写真の一番上に見え，制御論理はそのすぐ下に見えます．MMU TLBは右上角にあり，八つの1Kバイトキャッシュ・ブロックがチップの下部領域を占めています．

ARM810のおもな特徴を表12.3にまとめます．



〔図12.6〕ARM810のダイ写真

〔表12.3〕ARM810の特徴

プロセス	0.5 μm	トランジスタ数	836,022	MIPS	86
メタル層	3	ダイ面積	76 mm^2	消費電力	500 mW
V_{dd}	3.3 V	クロック	0–72 MHz	MIPS/W	172

12.3 StrongARM SA-110

StrongARM CPUは，ARM LimitedとDigital Equipment Corporation注が共同で開発しました．これは，修正ハーバード・アーキテクチャ（命令キャッシュとデータ・キャッシュが別々）を使用した最初のARMプロセッサでした．（注：DEC，現コンパックの一部門，その後StrongARM部門は，米国Intel社に買収された）

SA-110は現在，1998年にDigital Semiconductor社を買収したIntel社が製造しています．

◆Digital Alphaの背景

DECは，高クロック速度を誇る64ビットRISCプロセッサ「Alphaマイクロプロセッサ」を開発したことでマイクロプロセッサ業界で知られています．クロック周波数を高く保つ能力は，高度なCMOSプロセス技術，入念にバランスをとったパイプライン設計，完璧なクロッキング方式，これらの要素すべてを非常によく制御する自社設計ツールの賜物です．

StrongARMの設計においても同じアプローチが用いられ，そのうえ，高い電力効率を達成するという目的が加えられました．

◆StrongARMの構成

StrongARMの構成は，図12.7のとおりです．おもな特徴は次のとおりです．

(1) レジスタ・フォワーディング付き5段パイプライン

(2) 64ビット乗算，複数レジスタ転送，メモリとレジスタ用のスワップ命令を除く，一般的な全命令の単一サイクル実行

(3) 32バイト・ラインをもつ16Kバイト32ウェイ・アソシアティブ命令キャッシュ

(4) 32バイト・ラインをもつ16Kバイト32ウェイ・アソシアティブコピーバック・データ・キャッシュ

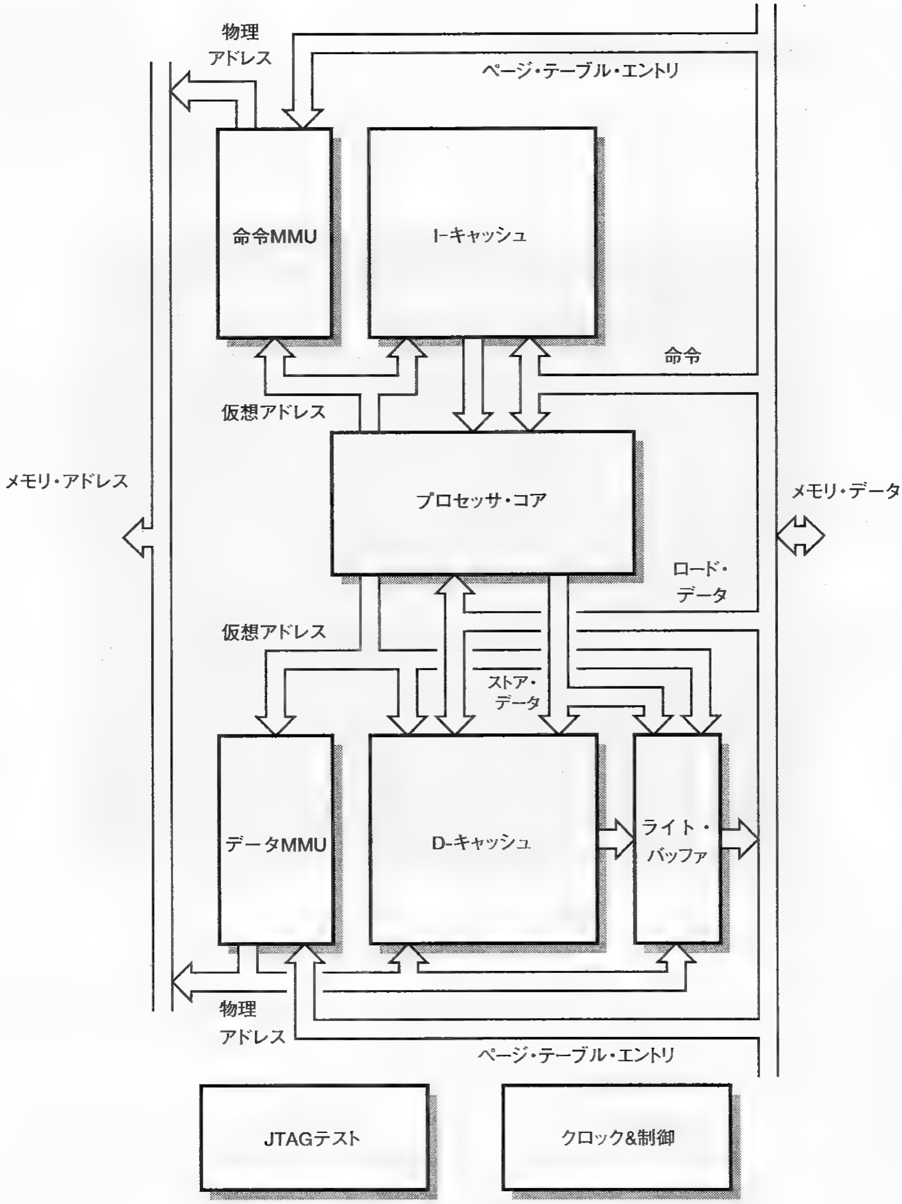
(5) 32エントリの命令変換ルックアサイド・バッファと32エントリのデータ変換ルックアサイド・バッファ

(6) エントリごとに最大16バイトまで入る8エントリのライト・バッファ

(7) 低消費電力の擬似スタティック動作

このプロセッサは，システム制御コプロセッサ15を使用してオンチップMMUとキャッシュ資源を管理し，プリント回路基板接続をテストするJTAGバウンダリ・スキャン・テスト回路を組み込んでいます（デバイス自体のインサーキット・テストに使うJTAG「インテスト」はサポートしていない）．

初代のStrongARMチップは，DECの0.35 μm CMOSプロセスで3層の金属層を用いて実装されました．チップは50mm^{2}ダイ上に（この性能をもつプロセッサにしては非常に小型）250万個のトランジスタを使用し，160から200MHzクロックで200から250 Dhrystone MIPSを出しました．また，消費電力は1.65Vから2Vの電源で0.5～1W未満でした．

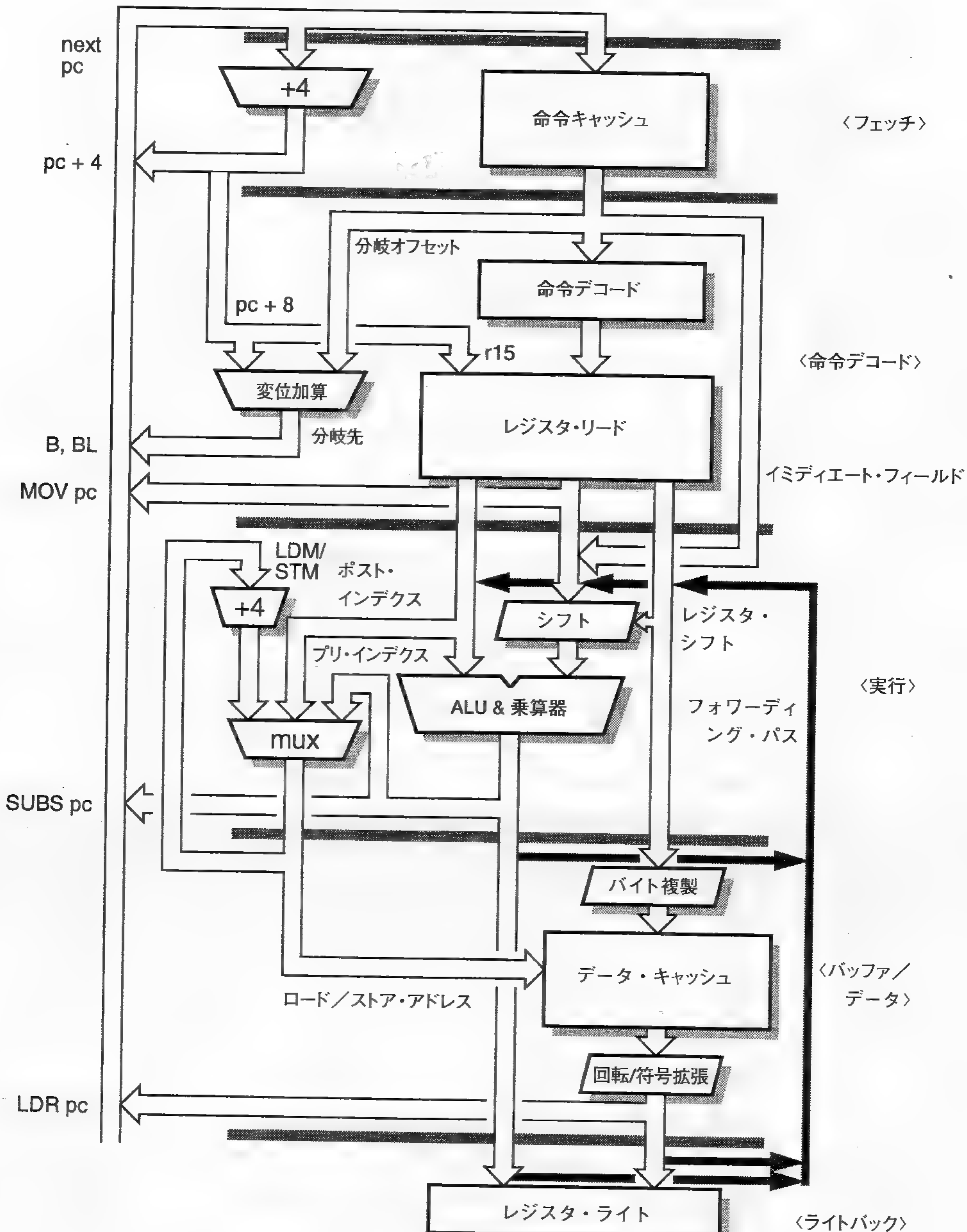


〔図12.7〕StrongARMの構成

◆StrongARMプロセッサ・コア

プロセッサ・コアは，フルバイパス(レジスタ・フォーワディング)およびハードウェア・インタロックを備えた古典的な5段パイプラインを採用しています．ARM命令セットでは，レジスタ・バンク読み出しアクセスの開始前にいくつかの命令デコードが必要で，また，ALUとシリーズにシフト動作も必要ですが，これら両方の論理機能は各パイプライン・ステージ内に収まっており，パイプライン段数は増えません．パイプライン・ステージは次のとおりです．

(1) 命令フェッチ（命令キャッシュから）
(2) 命令デコードとレジスタ読み出し．分岐ターゲット計算と実行
(3) データ転送メモリアドレス計算を含むシフトおよびALU処理
(4) データ・キャッシュ・アクセス
(5) レジスタ・ファイルへの結果ライトバック



〔図12.8〕StrongARMコア・パイプライン構成

おもなパイプライン・コンポーネントの構成を図12.8に示します．黒い太線はパイプライン・ステージの区切りで，これらのラインを超えるデータは交点でラッチされます．ラインをまたがずに供給するデータは，パイプライン・ステージ間でデータを戻したり先送りします．次に例をあげます．

(1) レジスタ転送パス．中間値を次の命令に渡し，リード・アフタ・ライト・ハザードによるレジスタ・インタロック停止を防ぐ．

(2) 次の命令のフェッチ・ステージからpc+4を転送するPCパス．pc+8を現在の命令に与え，r15として使用されたり分岐ターゲット計算に使用されたりする．

◆パイプラインの特徴

このパイプライン構造でとくに注目に値するのは次のことです．

(1) レジスタ制御シフトをイネーブルし，ベース＋インデクス・アドレシング付きストアを1サイクルで発行するための三つのレジスタ読み出しポート

(2) 自動インデクス付きロードを1サイクルで発行するための二つのレジスタ書き込みポート

(3) 複数ロードおよびストア命令をサポートする実行ステージのアドレス・インクリメンタ

(4) 次のPC(プログラム・カウンタ)値を得る多数のソース

最後の点は，ARMアーキテクチャではPCがr15としてレジスタ・バンク内で可視であるためさまざまな変更が可能であることを示しています．

◆PCの変更

次のPC(プログラム・カウンタ)を得るもっとも一般的なソースは，フェッチ・ステージのPCインクリメンタです．この値は次サイクルの最初で入手できるため，各サイクルで1命令をフェッチできます．

次に一般的なPCのソースは，分岐命令の結果です．専用分岐変位加算器は命令デコード・ステージの間にターゲット・アドレスを計算し，分岐そのものを実行するのに使うサイクルに加えて，実行した分岐が1サイクルのペナルティを起こします．オフセットは命令内の固定フィールドですから，変位加算は命令デコードと並行して起きることに注目してください．もし命令が分岐でないことがわかれば，計算したターゲットは単に廃棄します．

たとえばループの終了を制御する一般的なコード・シーケンスは，次のとおりです．

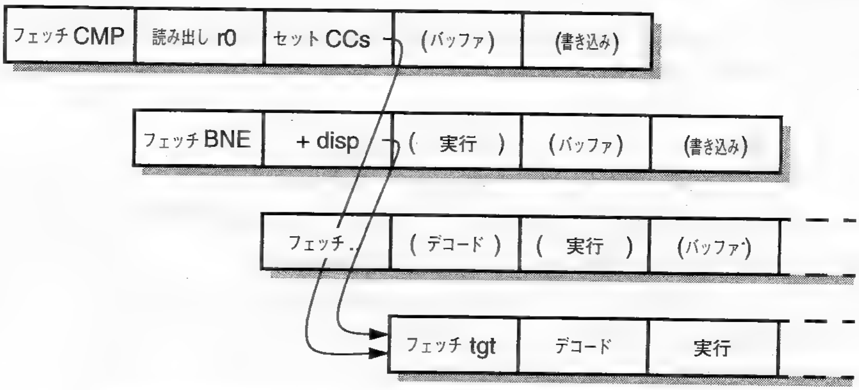
```
CMP     r0, #0
BNE     label
..
```

このシーケンス中のパイプライン処理は，図12.9のとおりです．分岐ペナルティが増加しないように，タイミングが間に合うように条件コードが有効になることに注目してください．分岐直後の命令はフェッチ，廃棄されますが，次にプロセッサは分岐ターゲットからフェッチを始めます．ターゲット・アドレスの生成と同時に条件コードも生成し，ターゲット・アドレスを使うかどうかを決定します．

同じ1サイクル・ペナルティは，リンク付き分岐命令の場合も起こります．この命令も同様に分岐をしますが，実行ステージと書き込みステージを使ってpc+4を計算し，r14，つまりリンク・レジスタに書き込みます．通常のサブルーチン復帰命令，MOV，pc，lrの場合も同様です．この場合，ターゲット・アドレスは分岐変位加算器ではなくレジスタ・ファイルからきますが，やはりデコード・ステージの終わりで入手できます．

たとえば例外から復帰する場合など，復帰アドレスを計算しなければならないときは，2サイクルのペナル

ティがつきます．なぜなら，ALU結果は実行ステージの終わりでしか入手できないからです．PCがメモリからロードされる（ジャンプ・テーブルから，またはスタックからのサブルーチン復帰）場合，ペナルティは3サイクルになります．



〔図12.9〕StrongARMループ・テスト・パイプライン動作

◆フォーワディング・パス

実行パイプラインは，各レジスタ・オペランドにつき三つのフォーワディング・パスがあり，リード・アフタ・ライトハザードが起こったときの停止を防ぎます．値が転送されてくるのは次の場所です．

(1) ALU結果
(2) データ・キャッシュからロードしてきたデータ
(3) バッファされたALU結果

これらのパスにより，ロードしたデータ値を次の命令が使用する場合を除き，データ依存の停止はすべてなくなります．次の場合においては，1サイクルの停止が必要です．

◆アボート回復

ALU結果をバッファして書き込みを最終ステージまで遅らせるより，次のステージでレジスタ・ファイルに書き込めば，パスのうち一つを省くことができるように思えるかもしれません．遅延方式にメリットがあるのは，データ・キャッシュ・アクセスの間にデータ・アボートが起こり，ベース・レジスタ値を回復するのに修復動作がいるかもしれないからです（たとえば，フォールトが起こる前にベース・レジスタが複数ロード・シーケンスで上書きされている場合）．この方式は，回復を可能にするだけでなく，もっともクリーンな回復メカニズムです．ベース・レジスタ値を命令開始時の値に，例外ハンドラが自動インデクスをundoする必要がなくなります．

◆乗算器

とくに注目に値するのは，StrongARMの乗算ユニットです．プロセッサの高クロック速度にも関わらず，12ビット/サイクルで計算し，1から3クロック・サイクルで二つの32ビット・オペランドの積を算出します．高速乗算器は，高いディジタル信号処理性能を要求するアプリケーションにおいてStrongARMの可能性を大きく伸ばすものです．

◆命令キャッシュ

命令キャッシュは，8命令（32バイト）の512ラインの中に16Kバイトの命令を保持します．キャッシュは32ウェイ・アソシアティブ（CAM-RAM構成を使用）で循環置換アルゴリズムをもち，プロセッサの仮想アドレスを用います．ブロック・サイズはライン・サイズと等しいため，ライン全体を一度にメモリからロードします．各メモリ領域は，メモリ管理テーブルを用いてキャッシュ可能またはキャッシュ不可とマークでき，キャッシュはソフトウェアの制御下でディセーブル，（完全に）フラッシュも可能です．

◆データ・キャッシュ

データ・キャッシュも命令キャッシュと同様の構成を用いますが，データ・ストア（命令は読み取り専用）に対応するための追加機能があります．容量は16Kバイトで，512の32バイト・ラインが循環置換付き32ウェイ仮想アドレス・アソシアティブ・キャッシュとして配列されています．ブロック・サイズは，32バイトです．キャッシュはソフトウェアによりディセーブルでき，各メモリ領域をキャッシュ不可にすることも可能です（たいていI/O領域はキャッシュ不可とマークするのが賢明です）．

データ・キャッシュはコピーバック書き込み方式を使用しています．ラインごとに二つのダーティ・ビットがあるので，ラインをキャッシュから削除すると，全部か半分をメモリに書き込むか，もしくはまったく書き込みません．一つでなく二つのダーティ・ビットを使用することで，メモリ・トラフィックを減らします．なぜなら「半ダーティ」な場合がきわめて多いからです．キャッシュは，ラインをメモリに再書き込みする際，各ラインと物理アドレスのコピーをストアし，削除されたラインをライト・バッファに入れます．

キャッシュがコピーバック書き込み方式を使用しているため，ダーティ・ラインすべてをメモリに再書き込みする必要がある場合があります．StrongARMでは，ソフトウェアを使って新しいデータを各ラインにロードし，ダーティ・ラインを削除します．

◆同義語

どんな仮想アドレス・キャッシュでも，キャッシュ可能な全物理メモリ位置が独自にマップした仮想アドレスをもつようにしなければなりません．同じ物理位置に二つの仮想アドレスがマップされると，仮想アドレスは同義語（synonyms）になります．同義語が存在すると，どちらの仮想アドレスもキャッシュできません．

◆キャッシュの整合性

命令キャッシュ，データ・キャッシュが分かれていると，同じ物理メモリ位置に関して矛盾したコピーをもつ可能性があります．メモリ領域を，あるときはデータ（書き込み可能）として扱い，またあるときは実行可能命令として扱う場合は，矛盾を防ぐために注意しなければなりません．

よくある例は，プログラムをロード（または，一つのメモリ位置から別の位置にコピー）し，その後実行する場合です．ロード段階では，プログラムはデータとして扱い，データ・キャッシュを通ります．そして実行される時は命令キャッシュ（同じアドレスからきた前の命令のコピーをもつこともある）にロードされるのです．正しく処理するには，次のようにします．

(1) ロード・フェーズは完了していなければならない．

(2) データ・キャッシュ全体を「クリーン」にする（上記のように新しいデータを各ラインにロードする）か，変更されたキャッシュ・ラインのアドレスがわかっている場合は，それらのラインを明示的にクリーンにし，フラッシュする．

(3) 命令キャッシュをフラッシュする（古い命令を削除する）．

代わりに，特定のメモリ領域をロードフェーズの間，キャッシュ不可にすることでもできます．

ARMコードによく見られるリテラル（命令ストリームに含まれるデータ項目）に対しては，この問題は起こりません．メモリ・ブロックは命令とリテラルの混合であるため両キャッシュにロードされますが，個別のワード（またはバイト）が一貫して命令もしくはデータのどちらかとして扱われているかぎり，問題は起こりません．プログラムがリテラルの値を変更することすら（ただし，めったにないし，良いことでもない），命令キャッシュにあるかもしれない命令の値に影響しない限り，許されるのです．しかし，リテラルそのものを避け，命令を含むコード領域とは別のデータ領域にデータを保存するほうが賢明です．

◆コンパイラに関して

定数を各ルーチンの終わりに置くのではなく，コンパイラ・ユニット間にまたがってプールするように置くコンパイラは，命令キャッシュとデータ・キャッシュに分かれていることを認識している必要があります．コンパイラは，これにより，データ・キャッシュに命令が混ざったり，命令キャッシュにデータが混ざったりするのを最小限に抑えます．

◆ライト・バッファ

ライト・バッファは，書き込みデータ帯域幅内の小さなピークをならし，メモリ・バスの飽和によるプロセッサ停止を防ぎます．バッファを一杯にする大きなピークは，プロセッサを停止させます．同じ16バイト領域に対する各書き込みはライト・バッファ内でマージされますが，バッファに書き込まれた最後のアドレスまでだけです．バッファは，書き込みと物理アドレスをマージして外部メモリをアドレスする際，仮想アドレスをコピーし，最大8のアドレス（各アドレスは16バイト境界でアラインされている）と，各アドレスに対してデータ16バイトをストアします（したがって，各アドレスはダーティな半ラインまたは単一の複数ストアから最大4レジスタを扱うことが可能）．

ライト・バッファは，ソフトウェアによってディセーブルでき，各メモリ領域はMMUページ・テーブルを用いてバッファ可能またはバッファ不可とマークできます．キャッシュ可能なすべての領域はバッファ可能です（置き換えられたキャッシュ・ラインはライト・バッファ経由で書き込まれる）が，キャッシュ不可領域はバッファ可能なこともバッファ不可なこともあります．通常，I/O領域はバッファできません．バッファなし書き込みの場合，ライト・バッファが空になるまで待ってから，メモリに書き込まれます．

データ・キャッシュでミスしたデータ読み出しは，ライト・バッファのエントリと比べてチェックし，整合性を確かめます．ただし，命令読み出しはライト・バッファと比較チェックしません．データとして使用したメモリ位置が命令として使用される場合は常に，特別な命令を使ってライト・バッファがドレーンしたことを確認します．

◆MMU構成

StrongARMは，標準的ARMメモリ管理アーキテクチャを採用しており，命令とデータに別々の変換ルックアサイド・バッファ（TLB）を使用しています．各TLBは，32変換エントリをもち，循環置換付きフルアソシアティブ・キャッシュになっています．TLBミスは，テーブル・ウォーキング・ハードウェアを起動し，主メモリから変換情報およびアクセス許可情報をフェッチします．

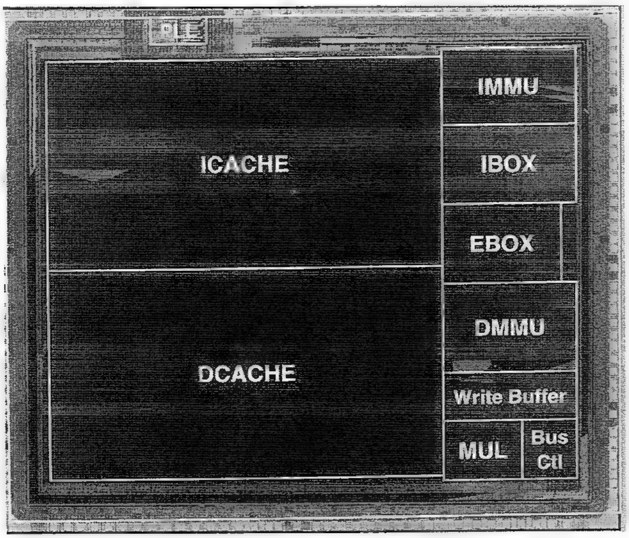
◆StrongARMシリコン

図12.10は，StrongARMダイの写真です．主要な機能領域が示してあります．ダイ領域は，当然ですが，

命令キャッシュ(ICACHE)とデータ・キャッシュ(DCACHE)が大半を占めています．各キャッシュには独自のMMU(IMMUとDMMU)があります．プロセッサ・コアには命令発行ユニット(IBOX)，実行ユニット(EBOX)，高速乗算ハードウェア(MUL)があります．そしてライト・バッファと外部バスコントローラでプロセッサ論理は完成です．

高周波数オンチップ・クロックは，フェーズ・ロック・ループ(PLL)が外部3.68MHzクロック入力から生成します．

StrongARMの特徴は表12.4にまとめました．



〔図12.10〕StrongARMのダイ写真

〔表12.4〕StrongARMの特徴

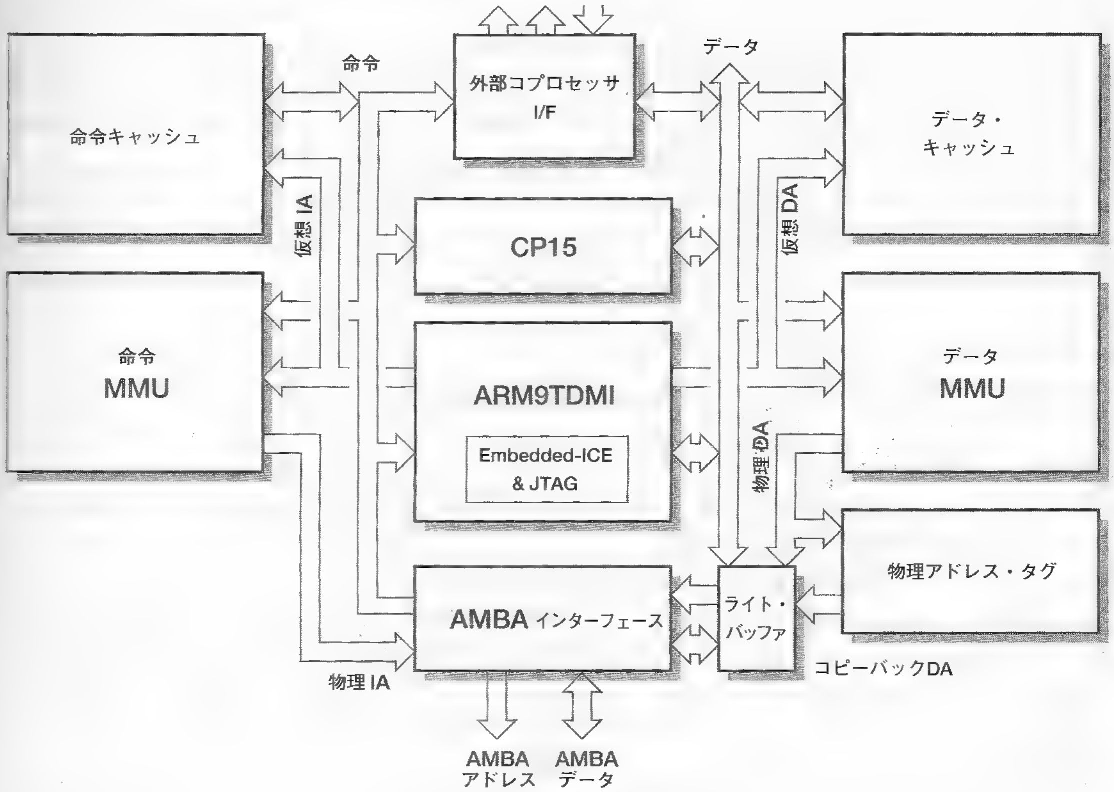
プロセス	0.35 μm	トランジスタ数	2,500,000	MIPS	115/268
メタル[illegible]	3	ダイ面積	50 mm^2	消費電力	300/1000 mW
V_{dd}	1.65/2 V	クロック	100/233 MHz	MIPS/W	380/268

12.4 ARM920TとARM940T

ARM920TとARM940Tは，ARM9TDMIプロセッサ・コア(9.3節参照)をベースとし，それに命令キャッシュとデータ・キャッシュが追加されています．命令ポートとデータ・ポートはAMBAバス・マスタ・ユニット経由で結合しており，ライト・バッファとメモリ管理ユニット(ARM920T)またはメモリ保護ユニット(ARM940T)も組み込まれています．

◆ARM920T

ARM920Tの全体構成は，図12.11に示したとおりです．



〔図12.11〕ARM920Tの構成

◆ARM920Tキャッシュ

命令キャッシュとデータ・キャッシュは両方ともサイズが16Kバイトで，セグメント化したCAM-RAM構成を使って64ウェイ・アソシアティブにしてあります．各キャッシュは64ラインの8セグメントで構成されており，必要なセグメントはA[7:5]でアドレスされます．セグメントは8ワード(32バイト)ラインで構成されており，256バイト単位(各セグメントの1ラインに対応)でのロックダウンをサポートしています．置換方式は擬似ランダムまたはラウンドロビンで，CP15のレジスタ1の「RR」ビット(ビット14)で指定します．キャッシュ・ミス時には8ワード・ライン全体を再ロードします．

命令キャッシュは読み出し専用です．データ・キャッシュはコピーバック書き込み方式をサポートしており，各ラインには一つの有効ビット，二つのダーティ・データ，一つのライトバック・ビットがあります．ライトバック・ビットは通常，変換システムに見られる情報を複製し，キャッシュがMMUを参照せずに書き込み動作をライトスルーもしくはコピーバックとして実現できるようにします．キャッシュ・ラインが置換されると全ダーティ・データがライト・バッファに送られ，その数は二つのダーティ・ビットにより0，4，または8ワードになります．データ・キャッシュは書き込みミス時ではなく読み出しミス時にのみスペースを割り当てます．

データ・キャッシュが仮想アドレスを使ってアクセスされると，ダーティ・ビットを主メモリにライトバックしなければならないたびに問題が起こります．なぜならこの動作には物理アドレスが必要だからです．ARM810は，仮想アドレスをMMUに返して変換することによってこの問題を解決していますが，必要な変換エントリがまだTLB内にあるという保証はもちろんありません．したがって，この処理全部が無駄に終わる可能性もあります．ARM920Tは，第2のタグ・ストアを使ってキャッシュ内の各ラインについて物理アドレスを保持することによって，この問題を解決しています．するとキャッシュ・ラインのフラッシュにMMUが関与する必要がなくなり，つねに遅れることなくダーティ・データをライト・バッファに転送することが可能となります．

ARM920Tは，キャッシュ・インデクスかメモリ・アドレスのいずれかを使ってダーティ・キャッシュ・ラインを主メモリに書き戻すことができます（「クリーニング」と呼ばれる処理）．したがって，特定のメモリ領域に対応する全エントリをクリーンにすることも可能です．

◆ARM920Tライト・バッファ

ライト・バッファは，最大4アドレスと16データ・ワードを保持します．

◆ARM920T MMU

MMUは，11.6節で説明したメモリ管理アーキテクチャを実装し，11.5節で述べたようにシステム制御コプロセッサCP15によって制御されています．ARM920Tは命令メモリ・ポートとデータ・メモリ・ポートを別々にもっていますから，MMUも各ポートに一つずつ計二つあります．

メモリ管理ハードウェアには，命令メモリ・ポート用の64エントリTLBとデータ・メモリ・ポート用の64エントリTLBがあります．ARM920Tは，WindowsCEをサポートするのに必要なProcessID論理を含んでいます．キャッシュとMMUはProcessID挿入の後にきますから，コンテキスト・スイッチがキャッシュやTLBを無効化することはありません．64Kバイトのラージ・ページと4Kバイトのスモール・ページのほか，ARM920TのMMUは1Kバイトのタイニー・ページの変換もサポートしています．

ARM920T MMUはTLBエントリに関して選択的ロックダウンをサポートしているため，リアルタイム処理などのクリティカルな変換エントリが追い出されないようになっています．

◆ARM920Tシリコン

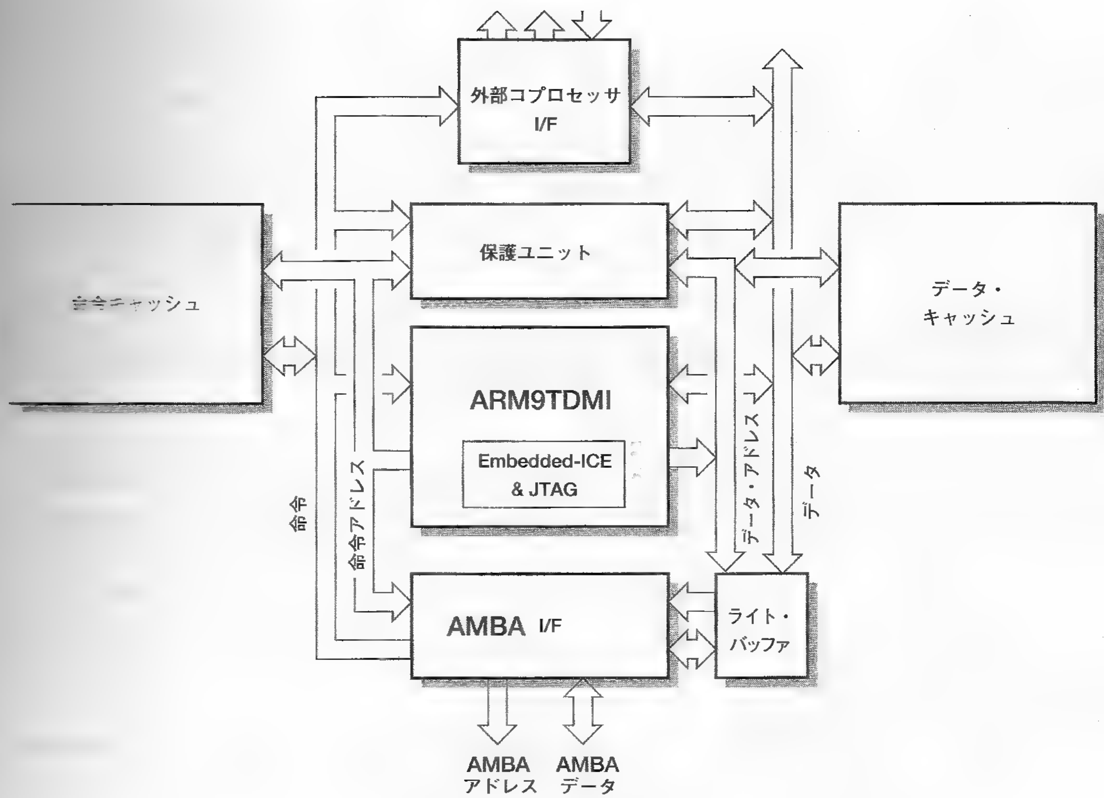
ARM920Tの特徴は表12.5にまとめました．

〔表12.5〕ARM920Tの特徴

プロセス	0.25 μm	トランジスタ数	2,500,000	MIPS	220
メタル層	4	コア面積	23–25 mm^2	消費電力	560 mW
V_{dd}	2.5 V	クロック	0–200 MHz	MIPS/W	390

◆ARM940T

ARM940TもARM9TDMIをベースとするCPUコアです．ARM940Tは仮想アドレスから物理アドレスへの変換をサポートしていないためARM920Tより単純です．ARM940Tの構成は，図12.12のとおりです．



〔図12.12〕ARM940Tの構成

◆メモリ保護ユニット

メモリ保護ユニットは，11.4節で説明したアーキテクチャを実装しています．ARM940Tは命令メモリ・ポートとデータ・メモリ・ポートを別々にもっていますから，メモリ保護ユニットも各ポートに一つずつ計二つあります．

この構成には，仮想アドレスから物理アドレスへの変換メカニズムがありません．多くの組み込み型システムにアドレス変換は必要ありませんし，完全なMMUはかなりのシリコン領域を必要とするので，必要でなければこれを省略することでかなりのコストが節約できます．AMBAインターフェースとアドレス変換ハードウェアの省略はどちらも，アプリケーションが組み込み型システムのものであり，同じチップ上に他のAMBAマクロセルがあることを想定しています．

◆ARM940Tキャッシュ

命令キャッシュもデータ・キャッシュもサイズは4Kバイトであり，1Kバイトセグメント四つで構成されています．各セグメントはフルアソシアティブCAM-RAM構造を使用しています（これらのキャッシュ用語については10.3節を参照）．両キャッシュは4ワード・ライン構造をもち，メモリ保護ユニットによってアドレスがキャッシュ可能と表示されている場合，キャッシュ・ミス時に常にライン全体をロードします．

キャッシュはロックダウンをサポートしているため，キャッシュを部分的にロードし，内容がフラッシュされ

ないよう保護することができます．ロックダウンされた部分のキャッシュは当然，もはや汎用キャッシュとしては使えませんが，一部のクリティカルなコード部分が常に確実にキャッシュにあるようにするほうがキャッシュのヒット率を最高にするより重要である場合もあります．キャッシュは16ワード単位でロックダウンできます．

ARM9TDMI命令ポートは命令をロードするために使用するため，読み出し動作しか実行しません．ただし，キャッシュがコピーをもつプログラムが主メモリで変更された場合には，整合性の問題が起こってきます．どんな変更をコードに加えるときも（いらなくなったプログラムが入っていたメモリに新しいプログラムをロードするなど）よく注意して，新しいプログラムを実行する前に命令キャッシュを選択的または完全にフラッシュしてください．

ARM9TDMIデータ・ポートは読み出しだけでなく書き込みもサポートしているため，データ・キャッシュは何らかの書き込み方式を実装していなければなりません（「書き込み方法」参照）．

ARM9TDMIは速いクロック速度で動作するため，データ・キャッシュはライトバック動作をサポートするよう設計されています．また単純なライトスルーも使えます．特定のアドレスに対してどのモードを選択するかは，メモリ保護ユニットが定義します．キャッシュ・ラインは読み出しミス時にのみ割り当てられます．

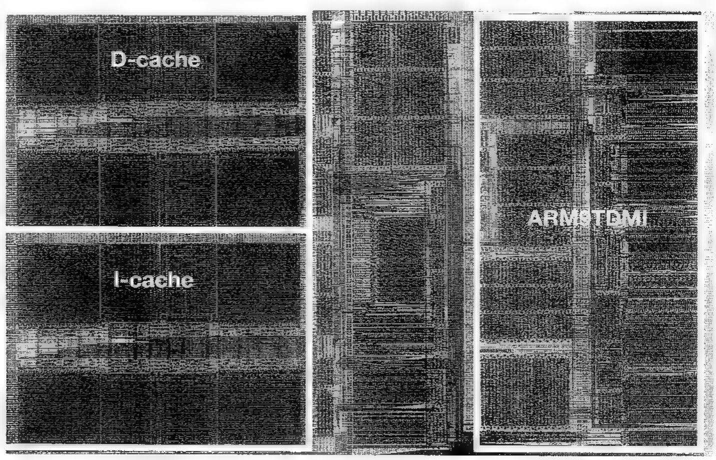
ARM940Tはソフトウェア・メカニズムを通じてキャッシュの「クリーニング」をサポートしており，各キャッシュ・ラインを確認してダーティなものがあればフラッシュします．当然，これには多少の時間がかかります．ARM920Tのようにメモリ・アドレスによってラインをクリーンするメカニズムはありません．

◆ARM940Tライト・バッファ

ライト・バッファは最大データ8ワードと4アドレスを保持できます．どのアドレスがバッファ可能であるかは，メモリ保護ユニットが定義しています．データ・キャッシュからフラッシュされたダーティ・ラインもライト・バッファを通過します．

◆ARM940Tシリコン

0.25μm CMOSプロセス上に実装されたARM940Tの特徴は表12.6にまとめてあります．また主な機能領域は，図12.13のとおりです．CPUコア全体は，低価格なシステムオンチップ（SOC）設計において比較的小さな領域しか占めていないことがわかります．



〔図12.13〕
ARM940T CPUコアのダイ写真

〔表12.6〕ARM940Tの特徴

プロセス	0.25 μm	トランジスタ数	802,000	MIPS	220
メタル層	3	コア面積	8.1 mm^2	消費電力	385 mW
V_{dd}	2.5 V	クロック	0–200 MHz	MIPS/W	570

12.5 ARM946E-SとARM966E-S

ARM946E-SとARM966E-Sは，ARM9E-S整数コアをベースとした合成可能CPUコアです(「ARM9E-S」参照).

両CPUコアにはAMBA AHBインターフェースがあり，8.8節で述べたとおり組み込み型トレース・マクロセルと合成することができます．これらのコアは組み込み型アプリケーション用ですから，アドレス変換ハードウェアはありません．

◆ARM946E-Sキャッシュ

ARM946E-Sは4ウェイ・セット・アソシアティブ・キャッシュをもっています．ARM920TやARM940TのCAM-RAM構成ではなくセット・アソシアティブ構成を選択しているのは，標準ASICライブラリにある合成可能RAM構造を使ってセット・アソシアティブ構成を構築するのが簡単だからです．ほとんどの設計システムにとってCAM-RAMキャッシュ構成を合成するのはまだ困難なことなのです．

命令キャッシュとデータ・キャッシュのサイズは，それぞれ4Kバイトから64Kバイトのいくらでもよく，両キャッシュが同じサイズでなくてもかまいません．両キャッシュともに8ワード・ラインを使用，ロックダウンをサポートし，擬似ランダムまたはラウンドロビンをソフトウェアで選択可能な置換アルゴリズムを採用しています．書き込み方式もライトスルーとコピーバックをソフトウェア選択可能です．

ARM946E-Sはメモリ保護ユニットを組み込んでおり，全体構成はARM940Tに似ています(図12.12参照).

◆ARM966E-Sメモリ

ARM966E-Sはキャッシュを使用していません．その代わり合成されたマクロセルには，固定メモリ・アドレスを割り当てた密結合で動作するSRAMが組み込まれています．このメモリのサイズはさまざまです．一つのメモリはデータ・ポートのみに接続し，2番目のメモリは両ポートに接続しています．一般に2番目のメモリは命令ポートが使用していますが，データ・ポート・アクセスも以下の二つの理由で必要です．

(1) コードに組み込まれた定数(アドレスなど)は，データ・ポートを通じて読み出さなければならない．

(2) 命令メモリを使用する前に初期化するメカニズムが必要である．命令ポートは読み出し専用であるから，命令メモリの初期化には使用できない．

またARM966E-Sは，AMBA AHB帯域幅を最適に利用するためにライト・バッファを組み込み，オンチップ・コプロセッサの接続もサポートしています．

◆ソフトIP

これらの合成可能CPUコアは，新しいプロセス技術ですぐに再合成できる高性能プロセッサを求める強い

市場の声に応えるものです．これらは，ARMテクノロジーを出荷する通常の形である「ハード」マクロセルの代わりとなるものです．

12.6 ARM1020E

ARM1020E CPUは，9.4節で説明したARM10TDMIコアをベースとしています．そしてThumb命令セットと8.9節で述べた信号処理命令セット拡張を含むARMアーキテクチャ・バージョンversion5TEを実装しています．

ARM1020Eの構成は図12.11に示したARM920Tに非常によく似ていますが，キャッシュ・サイズとバス幅が違います．唯一の実質的な違いは（この図に示した詳細のレベルで），命令側とデータ側で二つのAMBA AHBバス・マスタ要求-許可ハンドシェイクを使用することです．ARM920Tでは，内部でこれらの調停を行うため一つの外部要求-許可インターフェースしかありません．

◆ARM1020Eキャッシュ

ARM1020Eは，32Kバイトの命令キャッシュと32Kバイトのデータ・キャッシュを組み込んでいます．両キャッシュは64ウェイ・アソシアティブで，セグメント化したCAM-RAM構造を使用しています．キャッシュのライン・サイズは32バイトです．両キャッシュとも擬似ランダムかラウンドロビン置換アルゴリズムのいずれかを使用し，ロックダウンをサポートしています．

ARM1020Eの帯域幅条件を満たすため，両キャッシュは64ビット・データ・バスをもっています．命令キャッシュは1サイクルに2命令を供給し，データ・キャッシュは複数ロードの各サイクル中にデータ2ワードを供給するか，複数ストアの各サイクル中にデータ2ワードを書き込むことができます．

命令キャッシュは読み出し専用です．データ・キャッシュはコピーバック書き込み方式を採用し，1ライン当たり有効ビット一つ，ダーティ・ビット一つ，ライトバック・ビット一つがあります．キャッシュ内で1ラインが置換されると，ダーティ・ビットの状態に応じてゼロまたは8ワードがメイン・メモリにライトバックされます．64ビット・メモリ・データ・バスのおかげで完全な8ワードの排出（キャストアウト）処理を4メモリ・サイクルで実装することができます．

ライトバック・ビットは通常，変換システムにある情報を複製し，キャッシュがMMUを参照せずに書き込み動作をライトスルーまたはコピーバックとして実装することを可能にします．

◆ヒットアンダーミス・バッファ

ARM1020Eは「ヒットアンダーミス」動作にも対応しています．つまり，あるデータ参照がキャッシュ・ミスを起こすと，キャッシュはミスしたデータを含むラインをフェッチする一方で，後続のデータ参照には対応し続けることができます（さらにミスしないかぎり）．

◆ARM1020Eライト・バッファ

ライト・バッファは八つのスロットで構成されており，各スロットは1アドレスと64ビット・データのダブルワードを保持することができます．キャッシュのキャストアウトを処理するには独立した4スロット・ライト・バッファを使用し，主ライト・バッファの中でキャストアウトとヒットアンダーミスライト・トラフィックとの間に衝突がないようにします．

◆ARM1020E MMU

メモリ管理システムは，各キャッシュに一つずつ，二つの64エントリ変換ルックアサイド・バッファに基づいています．TLBは選択的ロックダウンもサポートしており，クリティカルなリアルタイム・コード部分の変換エントリが排出されないようになっています．

◆ARM1020Eバス・インターフェース

外部メモリ・バス・インターフェースはAMBA AHB準拠(8.2節参照)です．ARM1020Eには，命令メモリとデータ・メモリに別々のAMBA AHBインターフェースがあります．両インターフェースは32ビット・アドレス・バス・インターフェースと64ビット片方向読み出し書き込みデータ・バス・インターフェースを共有していますが，それぞれバス・アービタに対して独自の要求を行います．

◆ARM1020Eシリコン

0.18μm CMOSで1.5Vで動作するARM1020Eの目標性能は，表12.7にまとめてあります．CPUの電力効率を改善するには供給電圧を最小1.2V落として動作させます．

〔表12.7〕ARM1020Eの特徴

プロセス	0.18 μm	トランジスタ数	7,000,000	MIPS	500
メタル層	5	コア面積	12 mm^2	消費電力	400 mW
V_{dd}	1.5 V	クロック	0–400 MHz	MIPS/W	1,250

◆ARM10200

ARM10200はARM1020E CPUコアをベースとした参照チップ設計で，以下が追加されています．

- VFP10ベクタ浮動小数点ユニット
- 高性能SDRAMインターフェース
- 高速CPUクロックを生成するフェーズロック・ループ回路

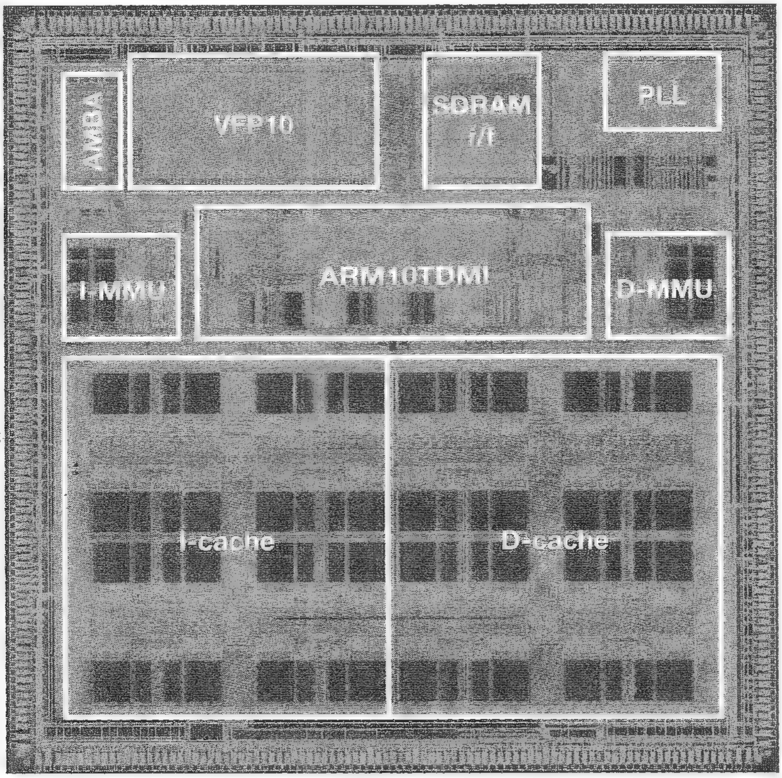
ARM10200は，ARM1020E CPUコアの評価用であり，ベンチマーク・テストやシステム・プロトタイピングをサポートするためのものです．

◆VFP10

ARM10TDMIは，付属のVFP10浮動小数点コプロセッサを通じて高性能ベクタ浮動小数点サポートを備えています．VFP10は，5段ロード/ストア・パイプラインと7段実行パイプラインを組み込み，単精度および倍精度IEEE754浮動小数点演算(6.3節参照)をサポートしています．また，クロック・サイクル当たり1回の速さで浮動小数点積和演算を発行することができます．そしてARM10TDMIの64ビット・データ・メモリ・インターフェースを利用して，1クロック・サイクルに倍精度値一つまたは単精度値二つをロードまたはストアします．演算命令もロード/ストア命令も一連のレジスタに対して同じ動作を実行する「ベクタ」形成を含み，ベクタ演算とベクタ・ロード/ストアは同時に動作することができますから，最大スループットは800MFLOPS(400 MHzで)となります．

◆ARM10200シリコン

0.25 mm ARM10200シリコンの主要な機能領域は，図12.14のとおりです．この最初のバージョンのチップは，完全に合成済みのVFP10コアを搭載し，キャッシュは一般的なデザイン・ルールにのっとって設計されています．0.18 mmバージョンのチップでは，VFP10コアにマニュアルでレイアウトされたカスタム・データパスと合成された制御が付き，キャッシュはもっとプロセスに固有のデザイン・ルールを使用して相対的なサイズを大幅に低減することになります．



〔図12.14〕ARM10200のダイ写真

12.7 考察

本章で説明したARM CPUコアは，高性能低消費電力プロセッサ・サブシステムの開発にとって多数の重要な側面にハイライトをあてています．

プロセッサ・コア自体の設計に関する問題は，第9章で論議し，9.5節でまとめました．ここでは，プロセッサ・コアに密接に関係し，コアが本来もっている性能を実際に生かすためにクリティカルなその他のコンポーネントを扱います．

◆メモリ帯域幅

プロセッサの性能は，最終的には接続したメモリ・システムの帯域幅によって制限を受けます．ARM CPUコア・ファミリは，キャッシュ・メモリ・システムがさまざまな性能レベルに最適化可能であることを示しています．

(1) ARM7TDMIは，メモリ・サイクル数をほとんど無駄にしないよう設計されています．これには，毎クロック・サイクルでデータ1ワードを供給できるメモリが必要です．フォンノイマン・アーキテクチャを採用しているため（つまり，一つのメモリ・ポートを命令転送とデータ転送の両方に使用する），統合キャッシュによってメモリ帯域幅条件を満たすことができます．ARM710T，720T，740Tはすべてそのようなキャッシュを搭載しています．

(2) ARM9TDMIは，クロック・サイクル当たり2ワード以上のデータを必要とするため，ハーバード・アーキテクチャを採用しています（命令メモリ・ポートとデータ・メモリ・ポートが別になっている）．そのニーズに応えるのは，別々のキャッシュ（とメモリ管理ユニット）です．ARM920Tと940Tは別々の32ビット・キャッシュを組み込んでいます．

(3) ARM10TDMIは，分岐が実行ユニットに入る前に予測して除去しようとするので，クロック・サイクル当たり二つ以上の命令を必要とします．また，複数ロード/ストア命令中にはサイクル当たりデータ2ワードを転送します．そのため，たとえばARM1020Eの64ビット・キャッシュのように，それぞれが32ビット以上の幅をもつ別々のキャッシュが必要です．

ARM9とARM10のCPUシリーズで使用する別々のキャッシュは，ソフトウェアで解決しなければならない整合性問題を伴うことがあります．必要な帯域幅をもたせる他の方法としては，ARM810の倍帯域幅キャッシュやAMULET3Hサブシステムのデュアル・ポート付きローカル・メモリ（14.6節で述べる）があります．これらの方法は両方とも統合メモリ・モデルを使用してソフトウェア設計を単純化していますが，ハードウェアはある程度複雑になります．

◆キャッシュの連想度

セット・アソシアティブRAM-RAMキャッシュとフルアソシアティブCAM-RAMキャッシュの相対的な長所については本章と第10章で十分に述べました．最も初期のARM CPU設計は，性能と電力効率の組み合わせを考慮してセグメント化したCAM-RAM構造を採用していました．ARM7シリーズのCPUで4ウェイ・セット・アソシアティブRAM-RAM構造に変更したのは，主にRAMがCAMより小さく（ビット当たりのレイアウト面積が半分ですむ），ARMが対応しなければならない多くのターゲット・プロセスの標準設計ライブラリで普通に入手可能だからです．後のARM8，ARM9，ARM10のCPUではセグメント化したCAM-RAM構造に戻りましたが，この理由の少なくとも一部はキャッシュのロックダウンのサポートを容易にするためです（例題12.1でさらに議論する）．

◆キャッシュ書き込み方式

ライトスルー・キャッシュは設計が最も単純で，プロセッサのクロック速度が主メモリのサイクル速度の数倍であればうまく動作します．ただしプロセッサのクロック速度がメモリ・サイクル速度の10倍以上にもなると，データ・ストア命令によって生成した書き込みデータ・トラフィックが外部メモリ・バスを飽和させ，書き込みサイクルが完了するのを待ってプロセッサがひんぱんにストールするようになります．この問題を避ける唯一の方法は，コピーバック・キャッシュを実装するなどして，全書き込みを主メモリに渡さないようなキャッシュ方式を使うことです．

現在の主メモリは通常，10MHz前後で動作しますから，ARM7シリーズのCPUはライトスルー・キャッシュを使って60MHzで十分に動作します．ARM810は最初の設計クロック速度ではボーダーラインですが，もしARM9シリーズに取って代わられていなかったら，もっとクロック速度が上がり，そのためにコピーバック・キャッシュ付きで設計されていたことでしょう．ARM9およびARM10のキャッシュは，性能を犠牲にしないとすれば，コピーバック方式を採用する必要があります．

◆キャッシュ・ラインの長さ

キャッシュ・ラインを長くすると，タグ・ストアのサイズが小さくなり(一定のデータ・ストア・サイズに対して)，キャッシュ・ミス率が低くなってミス・コスト(キャッシュ・ラインをロードするのにかかる時間)が大きくなります．ARM CPUコアはすべて，4ワード(16バイト)か8ワード(32バイト)ラインを使用しており，小型のキャッシュや組み込み型コードには小さいサイズ，汎用コードには大型のキャッシュが好まれています．ARM1020Eで使用されている64ビット・バスは，32ビット・バスが16バイト・ラインをロードおよびフラッシュするのにかけるのと同じメモリ・サイクル数で，32バイト・キャッシュ・ラインをロードおよびフラッシュすることができます．

◆メモリ管理

ARM MMUは，プロセッサ・コア自体と同じぐらいのシリコン面積を占める洗練された装置です．この機能性が必要であれば，搭載しておく必要があります．アプリケーション・コードの組み合わせが設計時にわからない汎用システムの場合は，たいていこの機能が必要です．既知のアプリケーション・コードを動作する組み込み型システムは，メモリ管理なしで動作するよう設計することができ，そのようなシステムでMMUのコストを正当化することは困難です．

ARM CPUコアは，汎用アプリケーション向けに完全なMMUを備えたシリーズと，決まったプログラムの組み込み型システム向けの単純なメモリ保護ユニットをもつシリーズに分かれて進化しました．ARM740TとARM940Tは後者の例であり，汎用CPUはWindowsCEをサポートするバージョン(ARM720T，ARM920T，ARM1020E)とサポートしないバージョン(ARM710TとARM810)を含みます．

メモリ管理アーキテクチャにおける最新の進歩は，ARM810，ARM920T，ARM1020EにおいてTLBロックダウンをサポートしたことです．これは，キャッシュ・ロックダウンと同様の役割を果たし，クリティカルなリアルタイム・コードをできるだけ効率的に動作させます．

12.8 例題と演習

＜例題12.1＞ セグメント化したアソシアティブ・キャッシュがセット・アソシアティブ・キャッシュよりロックダウンのサポートに適している理由は何か？

10.4節で説明したARM3などの初期のARM CPUのキャッシュは，CAM-RAM構造をもつ連想度の高いキャッシュを使用していました．ARM700シリーズのCPUはCAM-RAM構造を廃止し，RAM-RAMセット・アソシアティブ・キャッシュを採用しています．なぜならRAMタグ・ストアは，相当するCAMタグ・ストアよりダイ面積が小さくて済むからです．ARM810以降の新しいARM CPUは，CAM-RAMキャッシュに戻りました．少なくともこの理由の一つは，部分的キャッシュ・ロックダウンをサポートするためです．

CAM-RAMキャッシュがロックダウンのサポートに向いている理由は，単にその連想度レベルに関係して

います．特定のメモリ位置が対応する可能性のあるキャッシュ位置は，そのアドレス・ビットのいくつかをデコードすることで決定します．ダイレクト・マップ・キャッシュでは，キャッシュ位置は一つしかなく，2ウェイ・セット・アソシアティブ・キャッシュでは二つ，4ウェイでは四つの位置から選択可能です．

ロックダウンをサポートするARM CPUでは，ロックダウンはその選択肢の幅をせばめることによって動作します．4ウェイ・セット・アソシアティブ・キャッシュでは，キャッシュの4分の1をロックダウン（残りは3ウェイ・キャッシュ），キャッシュの半分をロックダウン（残りは2ウェイ），またはキャッシュの4分の3をロックダウン（残りは直接マップ・キャッシュ）するのが実用的です．これは粗い粒度であり，たとえば小さな割り込みハンドラを保持するメモリだけをロックダウンする必要がある場合は非効率的です．

CAM-RAMキャッシュは通常64ウェイ・セット・アソシアティブで，キャッシュは全体の64分の1単位でロックダウンできます．このほうがずっと粒度が細かくなります．

演習12.1.1 次のキャッシュ機能が性能と消費電力に与える影響を比較しなさい．

(1) タグおよびデータRAMを，並行ルックアップを実行する複数ブロックに分割し，連想度を上げる．
(2) タグおよびデータRAMアクセスをシリアルに行う．
(3) シーケンシャル・アクセスを用いてタグ・ルックアップをバイパスし，データRAMアクセス・メカニズムを最適化する．
(4) 別々に分かれたデータ・キャッシュ，命令キャッシュをもつ（StrongARM）．

演習12.1.2 なぜ，オンチップのライト・バッファはオフチップのメモリ管理ユニットと一緒に使えないかを説明しなさい．

演習12.1.3 なぜ，メモリ速度に対してプロセッサ速度が上がるにしたがって，書き込み方式をライトスルーよりコピーバックにしたほうがよいのですか？

第13章 組み込み型ARMアプリケーション

組み込みシステムの設計では，いくつかのメモリ・コンポーネントを除き，システムの主要機能を1個のチップに搭載する傾向が強まっています．コンポーネント・コスト，信頼性，電力効率における利点があるからです．この発展を支えるのは，半導体プロセス技術における進歩です．いまや何百万個ものトランジスタを組み込んだチップが安く生産でき，数年以内には何千万個ものトランジスタを組み込むことも可能になるでしょう．

このように，1チップ上に複雑なシステムが載る時代です．ARMはこの時代を先導する役割を果たしてきました．ARMの小さなコア・サイズにより，他のシステム機能にあてるシリコン・スペースが増えるからです．本章では，ARMベースのシステム・オン・チップ(SOC)の例をいくつか見ますが，これはこの世界のほんの表面に触れるだけにすぎません．数年後には，組み込み型アプリケーションがどっと市場にあふれ，その多くはARMプロセッサ・コアを中心としていることでしょう．

32ビット・コンピュータ・システムの設計は，複雑な仕事です．1チップ上に「right first time(最初から正しく)」ものを作るのは，非常に大きなチャレンジです．成功への王道はありませんが，設計者を助ける強力な設計ツールは数多くあります．いかなる工学技術でも同様ですが，他人の問題と経験を研究することによってしか前進はありません．既存の設計を理解することは，新しいものを創造するより簡単です．新しいSOCの開発は，ディジタル・エレクトロニクスにおける現在においての最大のチャレンジといってもいいでしょう．

13.1 VLSI Ruby II 高性能通信プロセッサ

VLSI Technology, Inc.は，ARMの最初の半導体パートナであり，Acorn Computers Limited，Apple Computer, Inc.とともにARM Limited独立に貢献しました．ARMとの関係はARM Limitedの誕生にまでさかのぼります．というのも，最初のARMプロセッサを1985年に製造し，1987年にAcorn Computersから技術ライセンスを取得しているからです．

VLSIは，Acorn Computersのために多数の標準ARMベース・チップを製造し，またApple Newtons向けARM610チップを生産しました．また，特定顧客向け製品のためのARMベース設計もいくつか生産し，その多くを標準部品として入手可能にしました．Ruby IIチップもそのような標準部品の一つで，ポータブル通信デバイスでの使用を意図しています．

◆Ruby IIの構成

Ruby IIの構成は図13.1のとおりです．チップはARMコアを中心とし，2Kバイトの高速(ゼロ・ウェイト・ステート)オンチップSRAMを含んでいます．クリティカル・ルーチンはアプリケーション制御のもとでRAMにロードすることにより，最高性能と最小消費電力を得ています．また，PCMCIAインターフェース，四つのバイト幅パラレル・インターフェース，二つのUARTなど，多数のピンを共有する周辺モジュール・セットがあります．モード選択ブロックが，つねにいまどのインターフェースの組み合わせが可能かを制御しており，バイト幅FIFOバッファのおかげでプロセッサは，バイトが転送されるたびに応答する必要がありません．

同期通信コントローラ・モジュールが多様な標準シリアル通信プロトコルに対応し，シリアル・コントローラ・モジュールがソフトウェア制御データ・ポートを提供しています．このデータ・ポートを使用することにより，Philipsにより定義されたI^2Cバスなど多様なシリアル制御プロトコルが実現可能です．このバスは，バッテリ・バックアップRAM，リアルタイム・クロック，E^2PROM，オーディオ・コーデックなど幅広いシリアル・デバイスの接続を可能にします．

外部バス・インターフェースは，8，16，32ビットのデータ・バスをもつデバイスに対応し，柔軟にウェイト状態を生成します．カウンタ・タイマ・ブロックには24ビット・プリスケーラに接続した三つの8ビット・カウンタがあり，割り込みコントローラがすべてのオンチップ，オフチップ割り込みソースのプログラム可能な制御を行います．チップには，次の四つの電力管理モードがあります．

(1) オン・ライン

全回路がフルスピードで動作します．

(2) コマンド

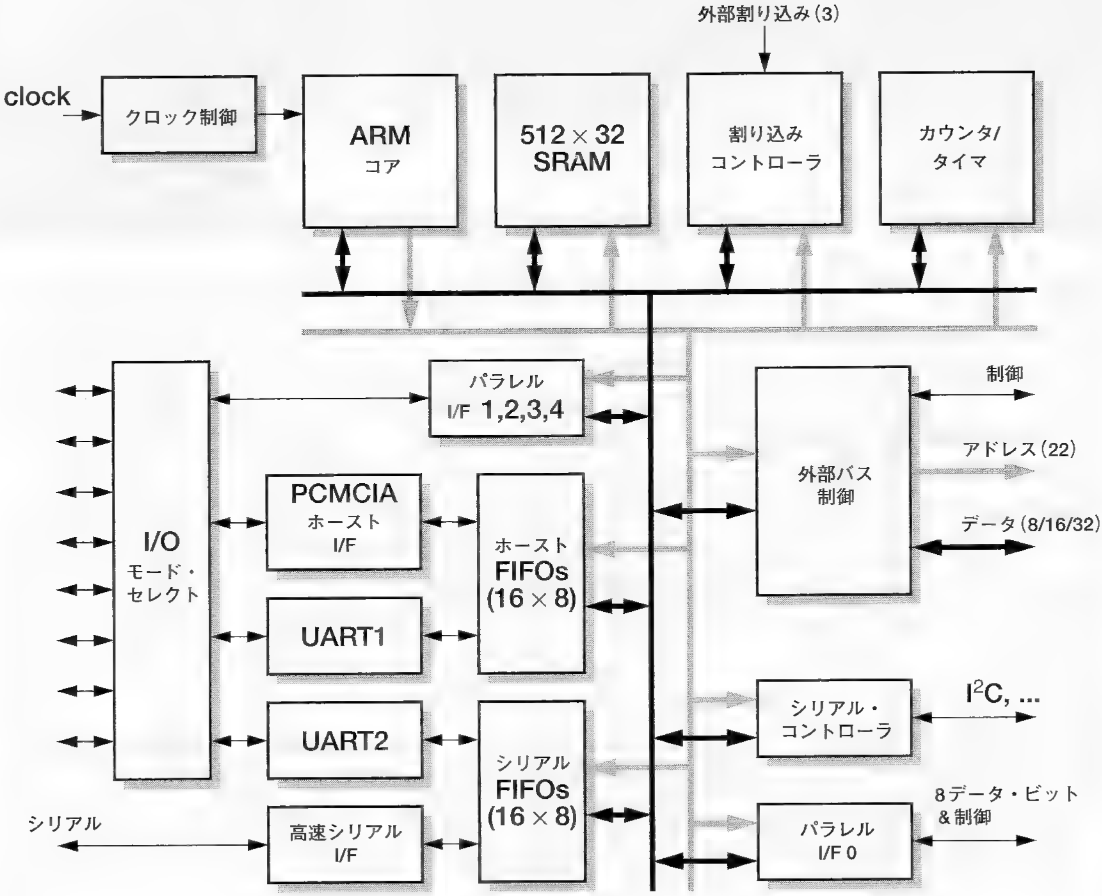
ARMコアには1から64のウェイト・ステート付きで動作し，その他の回路はすべてフルスピードで動作します．割り込みがあると，システムは即座にオンライン・モードにスイッチします．

(3) スリープ

タイマとオシレータを除く全回路が停止します．特定の割り込みによりシステムがオンライン・モードに戻ります．

(4) ストップ

全回路(オシレータを含む)が停止します．特定の割り込みによりシステムがオンライン・モードに戻ります．



〔図13.1〕Ruby Ⅱ高性能通信コントローラの内部構成

◆パッケージ

Ruby Ⅱは，144ピン，176ピンの薄型QFPで入手可能で，5Vで32MHzまで動作します．20MHzで32ビット・メモリを1ウェイ・ステートで使用した場合，チップはオンライン・モードで30mA，コマンド・モードで7.9mA，スリープ・モードで1.5mA，ストップ・モードで150μAを消費します．

13.2　VLSI ISDN Subscriber Processor

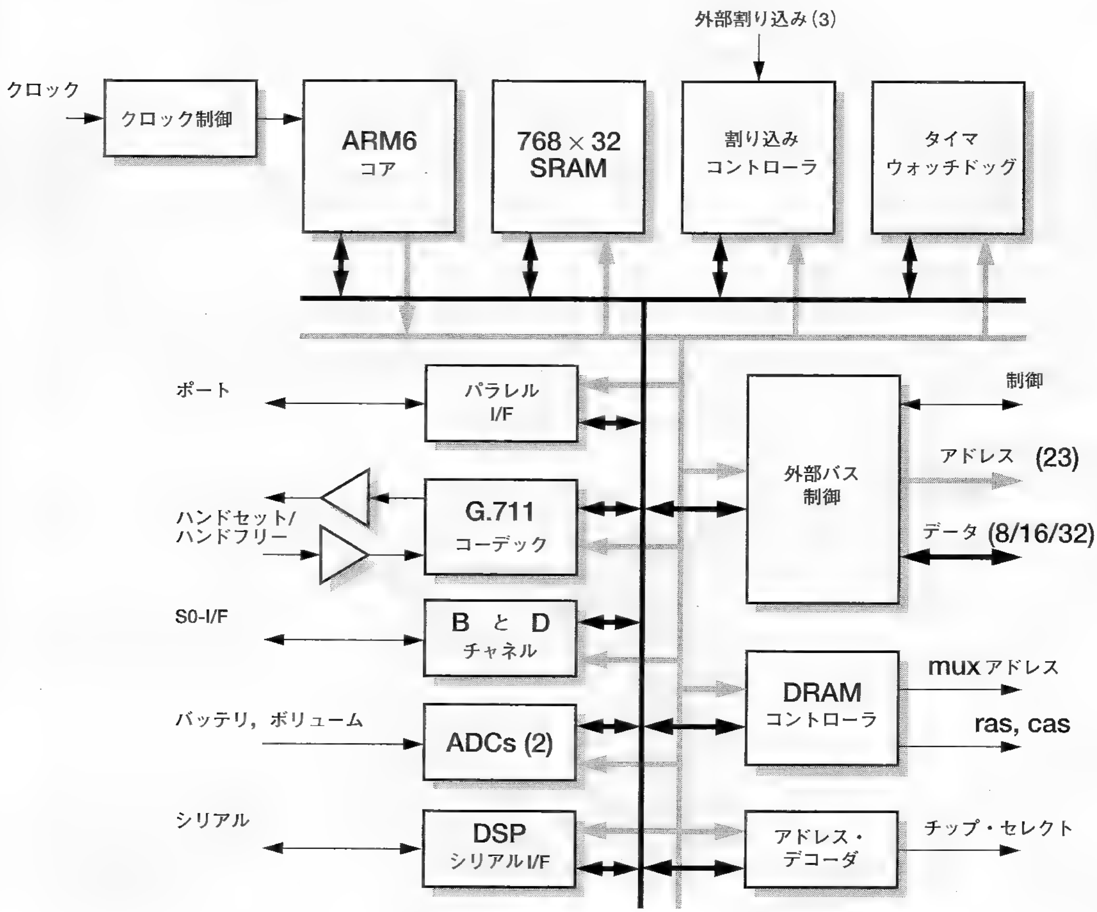
VLSI ISDN Subscriber Processor (VIP) は，「ISDN」(Integrated Services Digital Network．ディジタルテレフォニー規格) 加入者通信のためのプログラム可能なエンジンです．Hagenuk Gmbhが自社のISDN製品群向けに開発し，その後，「ASSP」(Application Specific Standard Part) として販売するためVLSI Technologyにライセンスを戻しました．これには，音声，データ，ビデオ・サービスを同一のディジタル回線でサポートするのに必要な機能をもつISDNターミナルを実現するのに必要なほとんどの回路が組み込まれています．

ターゲットとするアプリケーションには次のようなものがあります．

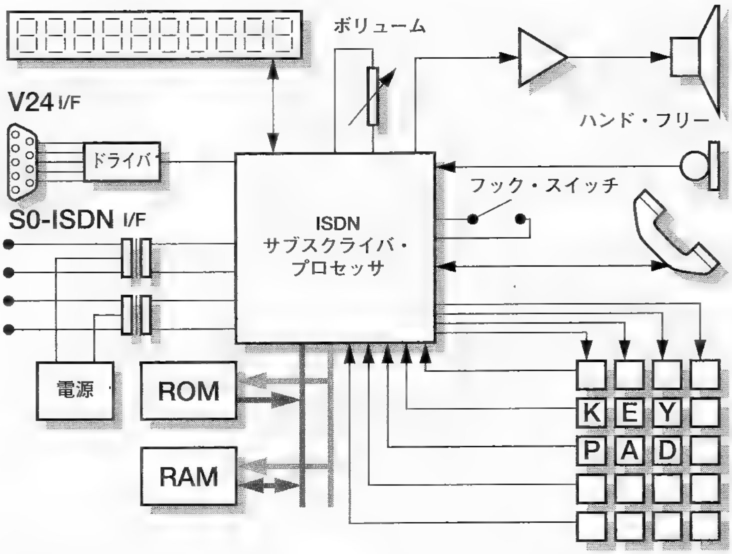
(1) ISDNターミナル装置．家庭用電話，ディジタルPABX電話，H.320ビデオ電話，統合PC通信など．
(2) ISDNから**DECT**(Digital European Cordless Telephone)へのコントローラ．多数のコードレス電話を相互に，またISDN回線にリンクする．家庭用およびビジネス用．
(3) ISDNからPCMCIAへの通信カード．

VIPチップは，特別なインターフェースを組み込んでおり，ISDN S0インターフェースに接続するほか，数字キーパッド，数字ディスプレイ，マイクなどの電話用インターフェースおよびイヤホン，外部信号プロセッサやコーデックへのディジタル・リンク，また，プログラマブル・クロックやバッテリ状態を監視するためのA-Dコンバータなどの電力管理機能をサポートします．

ARM6コアは，一般的な制御とISDNプロトコル機能を実行します．3KバイトのオンチップRAMは，ウェイト・ステートなしで36.864MHzの最大プロセッサ・クロック速度で動作します．ハンズ・フリー動作をサポートするのに必要な信号処理ルーチンなど，クリティカルなコード・ルーチンは必要に応じてこのRAMにロードします．



〔図13.2〕VIPの内部構成



〔図13.3〕VIPの典型的なシステム構成例

◆VIPの構成

VIPチップの構成は図13.2のとおりです．また，典型的なシステム構成は，図13.3のとおりです．

◆メモリ・インターフェース

外部メモリ・インターフェースは，8，16，32ビットのオフチップSRAMおよびROM，16，32ビットのDRAMをサポートしています．アドレス可能なメモリは四つの領域に分割され，それぞれがプログラム可能なウェイト・ステート数で動作します（最低は1ウェイト・ステートで，アクセス時間は54nsec）．

◆S0インターフェース

オンチップISDN S0インターフェースにより，絶縁トランスおよびサージ保護経由でS0インターフェース・バスへの接続が可能になります．オンチップ機能には，データおよびクロック復元のためのフェーズ・ロック・ループ，フレーミング，低レベル・プロトコルなどがあります．192kbpsのデータ転送レートには，二つの64kbpsのBチャネル，一つの16kbpsのDチャネルが含まれています．テレフォニ・アプリケーションでは，Bチャネルが8kHzのサンプリング・レートで8ビットの音声データを伝送し，Dチャネルが制御用に使用されます．

◆コーデック

G.711コーデックはオンチップ・アナログ・フロントエンドを含み，電話のハンドセット，ハンズ・フリー・マイクおよびスピーカの両方への直接接続を可能にしています．入力チャネルと出力チャネルの利得は，それぞれ独立してプログラム可能です．増幅段にはパワー・ダウンモードがあり，アクティブでない時は電力を節約します．

◆A-Dコンバータ(ADC)

オンチップのアナログ・ディジタル・コンバータは，コンデンサを入力電圧レベルにまで放電するのにかかる時間測定に基づいています．これは，ゆっくりと変化する電圧を測定するのに非常に簡単な方法で，オンチップ・コンパレータ，変換の最初でコンデンサを充電するための出力，変換の最初からコンパレータがスイッチする時点までの時間を測る手段ぐらいしか必要としません．典型的な用途としては，ボリュームを制御するポテンショメータからの電圧測定，携帯用途におけるバッテリ電圧チェックなどがあります．

◆キーパッド・インターフェース

キーパッド・インターフェースで使用するのは，キーパッドの列をストローブするパラレル出力ポート，行を感知する内部プルダウン抵抗付きパラレル入力ポートです．入力におけるORゲートは割り込みを生成することができます．列出力がすべてアクティブな場合，どんなキーを押しても割り込みを生成し，ARMは各列をアクティブにして各行を感知し，押したキーを特定することができます．

◆クロックとタイマ

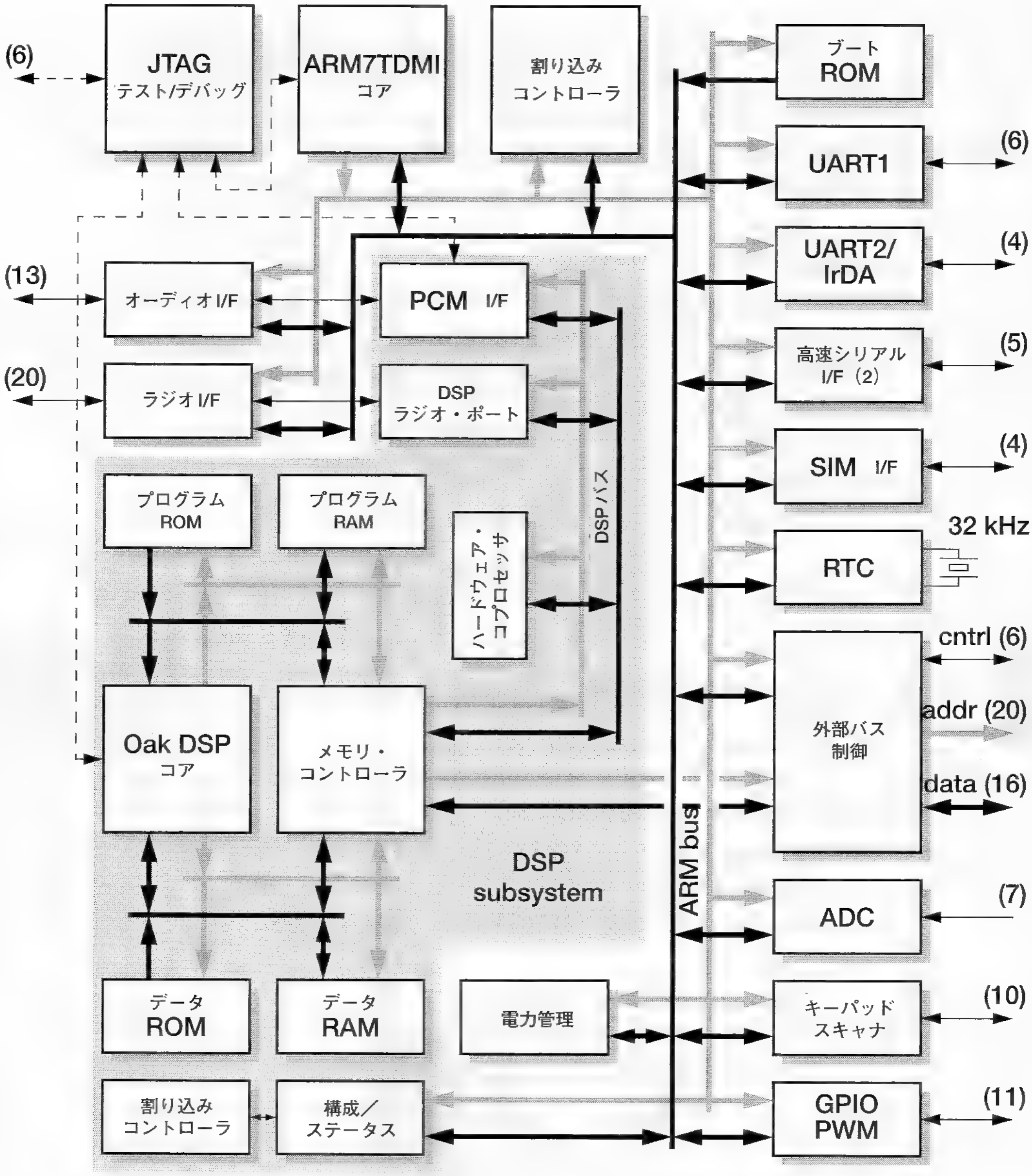
チップには二つのクロック・ソースがあります．通常動作は38.864MHzで，パワー・ダウン中は460.8kHzになります．ウォッチドッグ・タイマは，1.28秒間動作がないとCPUをリセットし，2.5msecタイマがDRAMのリフレッシュおよびマルチタスクのためにプロセッサのスリープ・モードに割り込みます．

13.3 OneC VW22100 GSMチップ

OneC VW22100は，GSM携帯電話ハンドセット用にVLSI Technologyが開発したシステムオンチップ設計です．OneC VW22100は，外部プログラムおよびデータ・メモリ，適切なラジオ・モジュールを追加すれば，ハンドセットに必要な全機能を備えます．OneC VW22100統合ベースバンド・デバイスを使用したGSMハンドセットの例としては，Samsung SGH2400があります．これは，ハンズ・フリー音声入力式ダイヤル機能を備えたデュアルバンド(GSM/1800)ハンドセットで，図13.4に写真があります．



〔図13.4〕
OneC VW22100を使用したサムソン社
SGH2400GSMハンドセット



〔図13.5〕OneC VW22100 GSMチップの内部構成

◆VW22100の構成

OneC VW22100のシステム・アーキテクチャは，現在の携帯電話ハンドセットに使用されているコントローラの典型です．このアーキテクチャでは，ARM7TDMIコアをユーザ・インターフェースとある程度のGSMプロトコル・レイヤを処理する汎用コントローラとして使用，DSPコアと特殊用途の信号処理ハードウェア・ブロックでベースバンド信号処理を行っています．このことは，図13.5に示します．

◆DSPサブシステム

DSPサブシステム(図13.5で影付きの部分)は，16ビットOak DSPコアをベースにしています．これは，以下を含むリアルタイム信号処理機能を全部実行します．

(1) ボイス・コーディング
(2) イコライゼーション
(3) チャネル・コーディング
(4) エコー・キャンセル
(5) ノイズ抑制
(6) 音声認識
(7) データ圧縮

◆ARM7TDMIサブシステム

ARM7TDMIコアは，以下を含むシステム制御機能を果たします．

(1) ユーザ・インターフェース・ソフトウェア
(2) GSMプロトコル・スタック
(3) 電力管理
(4) ペリフェラル・インターフェースの駆動
(5) 一部のデータ・アプリケーションの動作

◆役割分担

ARMコアとOakコアのタスク分割は，図13.5の詳細をよく調べるとはっきりします．オーディオ・インターフェースとラジオ・インターフェース(図の左側)は，両方の処理システムに接続しています．ARMシステムは増幅器のゲインを設定して，ラジオ伝送電力レベルと周波数シンセサイザを制御，リンガがユーザに着信を知らせるなどします．

ラジオ・リンクを通じて送信または受信されたエンコード済み音声データおよび記号を含むデータ・ストリームは，Oakシステムとペリフェラル・インターフェースの間で直接渡されます．

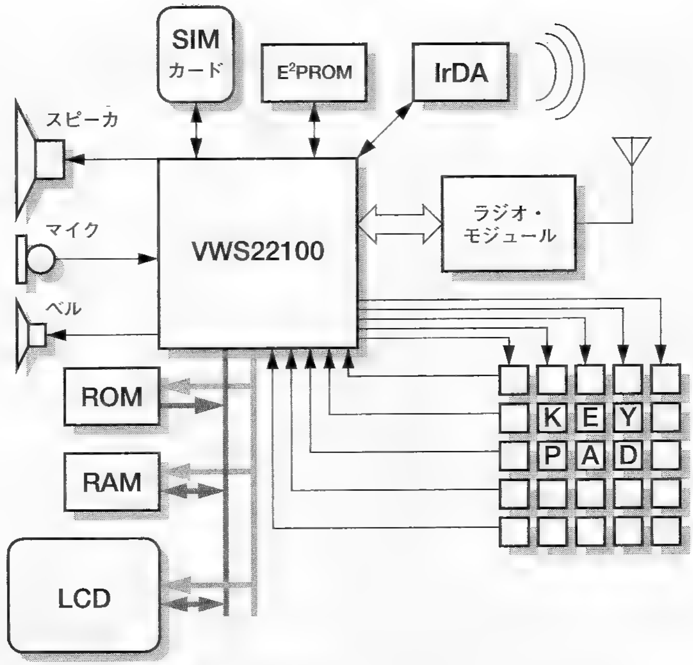
◆オンチップ・デバッグ

チップ上には二つの異質な処理システムがあります．オンチップ・デバッグ・ハードウェアは一つのJTAGインターフェースを使って，ARM7TDMI Embedded-ICEモジュール，Oak DSPコア上のデバッグ部，その他のテストおよびデバッグ機能など複数のデバッグ機能にアクセスします．

◆電力管理

携帯電話ハンドセットのバッテリ寿命は，現在の市場では重要な問題です．「通話時間」と「待受時間」は，製品広告で大きく強調されています．OneC VW22100は，以下のように多数の電力管理機能を組み込んで制御する製品の性能を最適化しています．

(1) 全体的/選択的パワーダウン・モード
(2) アイドル・モードでシステム・クロックの速度を低下
(3) アナログ回路も低電力で動作可能
(4) オンチップ・パルス幅変調出力がバッテリ充電を制御



〔図13.6〕典型的なGSMハンドセットのアーキテクチャ

(5) オンチップ・アナログ/ディジタル・コンバータ(ADC)が温度とバッテリ電圧をモニタして最適な動作を実現

◆GSMハンドセット

OneC VW22100をベースとしたGSM携帯電話ハンドセットの典型的なハードウェア・アーキテクチャは，図13.6のとおりです．

13.4 Ericsson-VLSI Bluetoothベースバンド・コントローラ

Bluetoothは，Ericsson，IBM，Intel，Nokia，東芝を含む企業コンソーシアムが開発した2.4GHz帯向け無線データ通信の事実上の標準です．この標準は，現在IrDA標準を使った赤外線通信で行っているのと同様の近距離通信(10cmから10mの範囲)をサポートするためのものですが，IrDAにあるラインオブサイト(視界)，アライメント(光軸)，相互干渉の制約を回避しています．Bluetoothはラジオ通信を使用して，ラップトップと携帯電話，プリンタ，PDA，デスクトップ，ファクス機，キーボードなどを結び，既存のデータ・ネットワークとのブリッジも提供されています．そのためパーソナル・ネットワークにおいてケーブルに代わる技術となります．

Bluetoothは，データ速度1Mビット/秒に対応し，周波数ホッピング方式とフォワード・エラー・コレクションの採用により，ノイズが多く未調整の環境でも安定した通信を実現します．

Ericsson-VLSI Bluetoothベースバンド・コントローラ・チップは，共同で開発された標準部品で，携帯用Bluetooth対応通信機器に使用するためのものです．

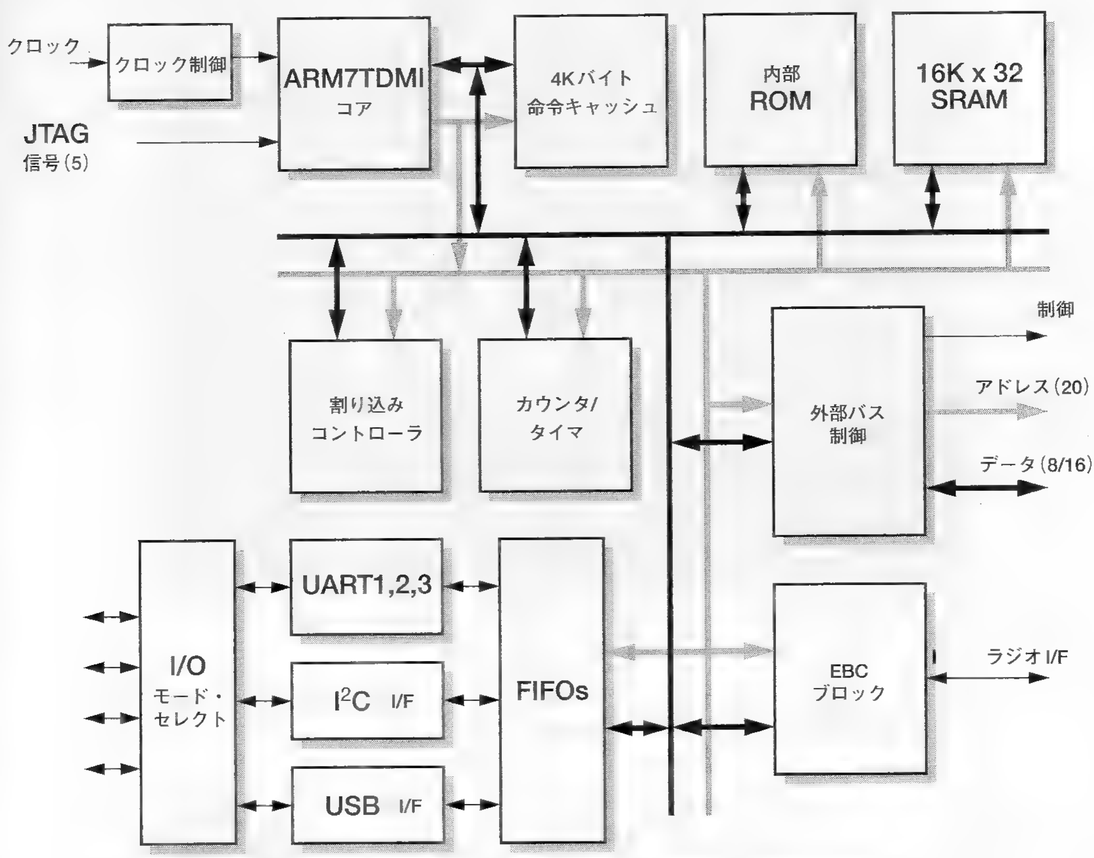
◆ Bluetooth「ピコネット」

Bluetooth対応装置は，臨時的に「ピコネット」を動的に形成します．これは，同じ周波数ホッピング方式で動作する2台から8台の装置群です．装置はすべて全く同じ実装をもつ同等のスレーブであり，うち1台がピコネット形成時にマスタとして働きます．マスタは，クロックおよびホッピング・シーケンスを定義し，それがピコネットを同期化します．

複数のピコネットがリンクして「スキャッタネット」を形成することもあります．

◆ Bluetoothコントローラの構成

Bluetoothベースバンド・コントローラの構成は，図13.7のとおりです．チップは，合成されたARM7TDMIコアをベースとし，64Kバイトの高速（ゼロ・ウェイト・ステート）オンチップSRAMと4Kバイトの命令キャッシュを組み込んでいます．クリティカル・ルーチンは最高性能を得るためにRAMへロードすることができます．キャッシュは，オフチップ・メモリ上にあるコードの性能と電力効率を向上します．



〔図13.7〕エリクソンVLSIのBluetoothコントローラの内部構成

ペリフェラル・モジュール群は，三つのUART，USBインターフェース，I2Cバス・インターフェースなど多数のピンを共有しています．FIFOバッファにより，プロセッサはこれらのインターフェースを通じて転送されるバイトごとに応答する必要がありません．

外部バス・インターフェースは，8ビットと16ビットのデータバスをサポートし，柔軟なウェイト・ステートが生成できます．カウンタ・タイマ・ブロックには，24ビット・プリスケーラに接続した三つの8ビット・カウンタがあり，割り込みコントローラがオンチップ，オフチップ全部の割り込みソースを制御しています．

◆Ericsson Bluetoothコア

Bluetoothベースバンド・コントローラには，電力最適化済みのハードウェア・ブロックであるEricsson Bluetoothコア（EBC）が組み込まれており，Bluetooth仕様内のリンク・コントローラ機能全部を処理するとともにBluetoothラジオ実装部分へのインターフェース論理を含んでいます．EBCは，ポイントトゥポイント，マルチスロット，ポイントトゥマルチポイントの各通信におけるパケット処理機能すべてを実行します．

ベースバンド・プロトコルは回路およびパケット・スイッチングの組み合わせを使用しています．たとえば音声伝送に対応するためにスロットを同期チャネル用に，リザーブすることも可能です．

◆電力管理

チップには次の四つの電力管理モードがあります．

(1) オン・ライン

全ブロックが通常スピードで動作します．ARM7TDMIコアのクロックは13～40MHzで，アプリケーションによって決まります．最大データ転送速度における電流消費は約30mAです．

(2) コマンド

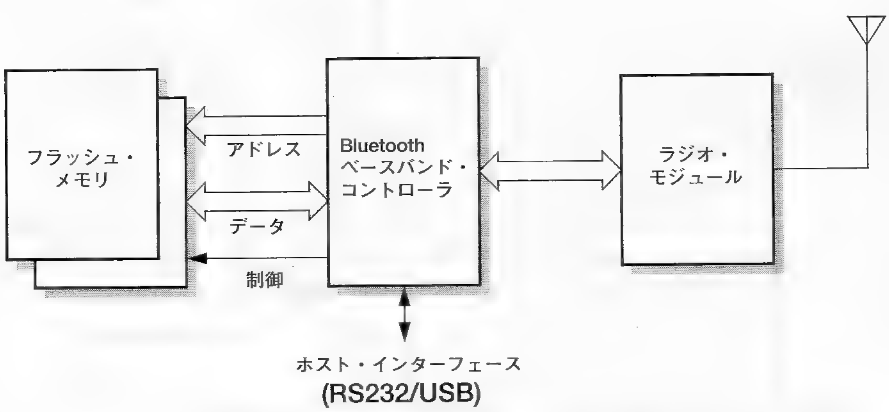
ARM7TDMIのクロックは，ウェイト・ステートを挿入することで遅くなります．

(3) スリープ

ARM7TDMIのクロックは停止し，その他ブロックのプログラム可能サブセットのクロックも停止します．このモードで使用する電流は，約0.3mAです．

(4) ストップ

クロック発振器が停止します．



〔図13.8〕典型的なBluetoothアプリケーション

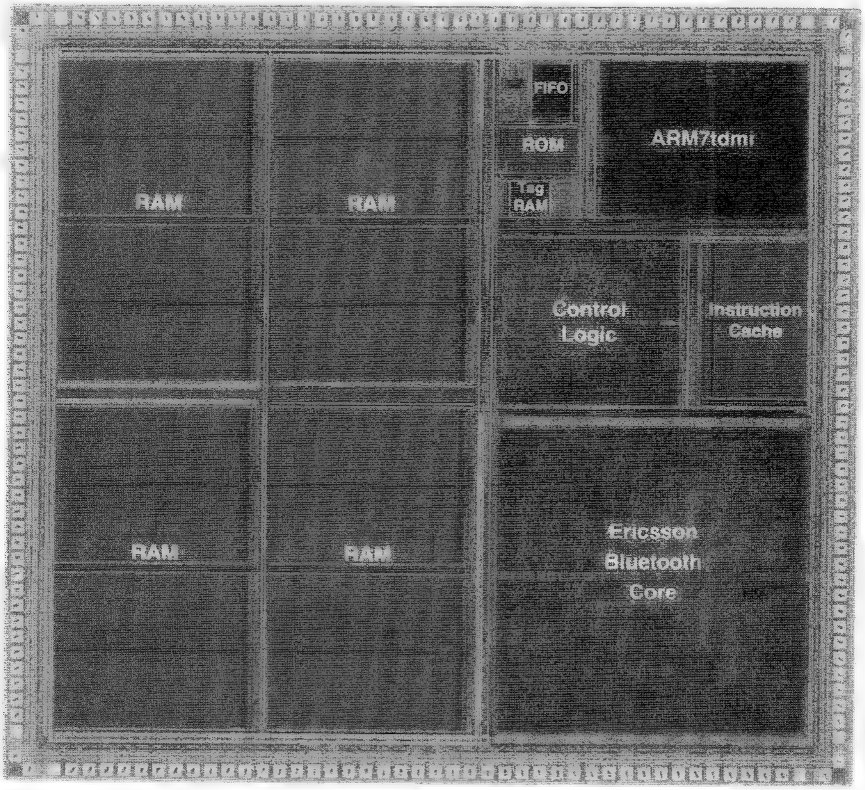
◆Bluetoothシステム

典型的なBluetoothシステムは，図13.8のとおりです．システムを完成させるには，ベースバンド・コントローラ・チップに外部ラジオ・モジュールとプログラムROMが必要です．高集積化により，高度で非常に機能的なラジオ通信システムが小型かつ経済的に実現されています．

◆Bluetoothシリコン

Bluetoothダイの写真は，図13.9のとおりです．ダイ面積の大半を占めているのが64KバイトのSRAM，次に大きいのが合成済みEBC（右下の4分の1）です．

合成されたARM7TDMIコアは右上角にありますが，図9.4のARM7TDMIハード・マクロセルに比べてずっと目立たない構造になっています．これは，このハード・マクロセルの場合は手作業でレイアウトされてお



〔図13.9〕Bluetoothベースバンド・コントローラのダイ写真

〔表13.1〕Bluetoothベースバンド・コントローラの特性

プロセス	0.25 μm	トランジスタ数	4,300,000	MIPS	12
メタル層	3	ダイ面積	20 mm^2	消費電力	75 mW
V_{dd}	2.5 V	クロック	0–13 MHz	MIPS/W	160

り，設計者が非常に規則的なデータパス構造を使って高密度なレイアウトにし，作らなければならないセルの種類をできるだけ少なくしたからです．合成済みセルは，もっと密度が低くさほど規則的な構造ではありません．合成済みコアの利点は，新しいCMOSプロセスに簡単に移植できることです．

Bluetoothコアの特徴は，表13.1にまとめてあります．このプロセッサは，最大39MHzで動作可能ですが，表に示したデータは典型的なGSMアプリケーションの例です．チップI/Oは3.3Vで動作しますが，コア論理は通常2.5Vで動作します．

◆ディジタル・ラジオ

安価で高性能なディジタル・システムの登場により，携帯電話市場の急成長に象徴されるように，ラジオ通信に新しいアプリケーション分野を開きました．ディジタル通信により，干渉によって起こるエラーのほとんどが検出，訂正可能な形でデータを送信できるようになりました．また，ディジタル技術によって周波数ホッピングのような高度な無線技術も制御可能となり，干渉の影響はさらに少なくなっています．

Bluetoothは，この利点をもっと小さな通信ネットワークにも適用し，個人だけでなくその人のもつ個人向けデータ機器全部が目に見えない形で互いに，そしてグローバルなネットワークに接続する将来を想定しています．

◆Bluetoothシステム・オンチップ

Bluetoothのアプリケーション分野は，ディジタル機能とラジオ機能を1個のCMOSチップに集積しようとする研究の焦点となりました．そうなれば図13.8のような典型的なアプリケーションが1個のチップ上に実装できます．

EBCブロックは，その他のAMBA搭載設計用にもIPとしてライセンス可能です．

13.5 ARM7500とARM7500FE

ARM7500は，高度に集積されたシングルチップ・コンピュータで，Acorn Risc PC（メモリ以外）の主要部品を単一チップ上に組み合わせたものです．プロセッサ・マクロセルとして，基本整数プロセッサ・コアだけでなくキャッシュやメモリ管理も含んだ完全なARM CPUを採用した最初のチップです．ARM7500FEは，オンチップ・マクロセルとしてFPA10浮動小数点コプロセッサを追加し，他に細かい変更を多数加えています．

ARM7500もARM7500FEも，セットトップ・ボックス，インターネット機器，ゲーム・コンソールなどの民生用マルチメディア・アプリケーションに適していますが，他にも多くのアプリケーション分野があります．

ARM7500とARM7500FEチップ上の主要マクロセルは次のとおりです．

(1) ARM CPUコア
(2) FPA10浮動小数点コプロセッサ（ARM7500FEのみ）
(3) ビデオおよびサウンド・マクロセル
(4) メモリおよびI/Oコントローラ

◆ARM CPUコア

ARM CPUコアは，ARM710 CPUの機能をほとんどすべて含んでいます．唯一の違いは，他マクロセルのための場所を空けるためにキャッシュを8Kバイトから4Kバイトにサイズ縮小したことです．

CPUは，ARM7整数コア(ARM7TDMIの先行モデルで，Thumbと組み込みデバッグ・サポートはない)を中心とし，4Kバイト4ウェイ・セット・アソシアティブで命令およびデータ混在キャッシュ，2レベル・ページ・テーブルに基づくメモリ管理ユニットと64エントリTLB，ライト・バッファをもっています．

◆FPA10浮動小数点ユニット

FPA10は，6.4節で多少説明しました．これは，システムが浮動小数点データ型(コプロセッサがない場合，ARM浮動小数点ライブラリ・ルーチンを使って処理する)を処理する能力を大幅に向上します．浮動小数点性能は，40MHzで最大6MFLOPS(倍精度浮動小数点コードを動作するLinpackベンチマークを使用して測定)になります．

◆ビデオおよびサウンド・マクロセル

ビデオ・コントローラは，最高120MHzのピクセル・クロック(オンチップの位相比較器を用い，オフチップのVCOで生成する)を使用して表示信号を生成します．256エントリ・カラー・パレットがあり，赤，緑，青の各出力に対応したオンチップ8ビットD-Aコンバータ，外部ミキシングやフェードのための追加制御ビットをもっています．ハードウェア・カーソルをサポートし，出力は高解像度カラー・モニタやシングルまたはダブル・パネルのグレー・スケールまたはカラー液晶ディスプレイを駆動できます．ディスプレイ・タイミングは完全にプログラム可能です．

ARM7500サウンド・システムは，独立した8チャネル8ビット(対数)アナログ・ステレオ・サウンドを生成し，オンチップのD-Aコンバータを通して音を出します．また，シリアルのディジタル・チャネルと外部CDクオリティD-Aコンバータを通して16ビット・サウンドを生成することもできます．ARM7500FEだけが後者の16ビット・サウンド・システムをサポートしています．

ビデオ，カーソル，サウンド・ストリームのためのデータ・チャネルは，メモリとI/Oのコントローラ内にあるDMAコントローラを使って生成されます．

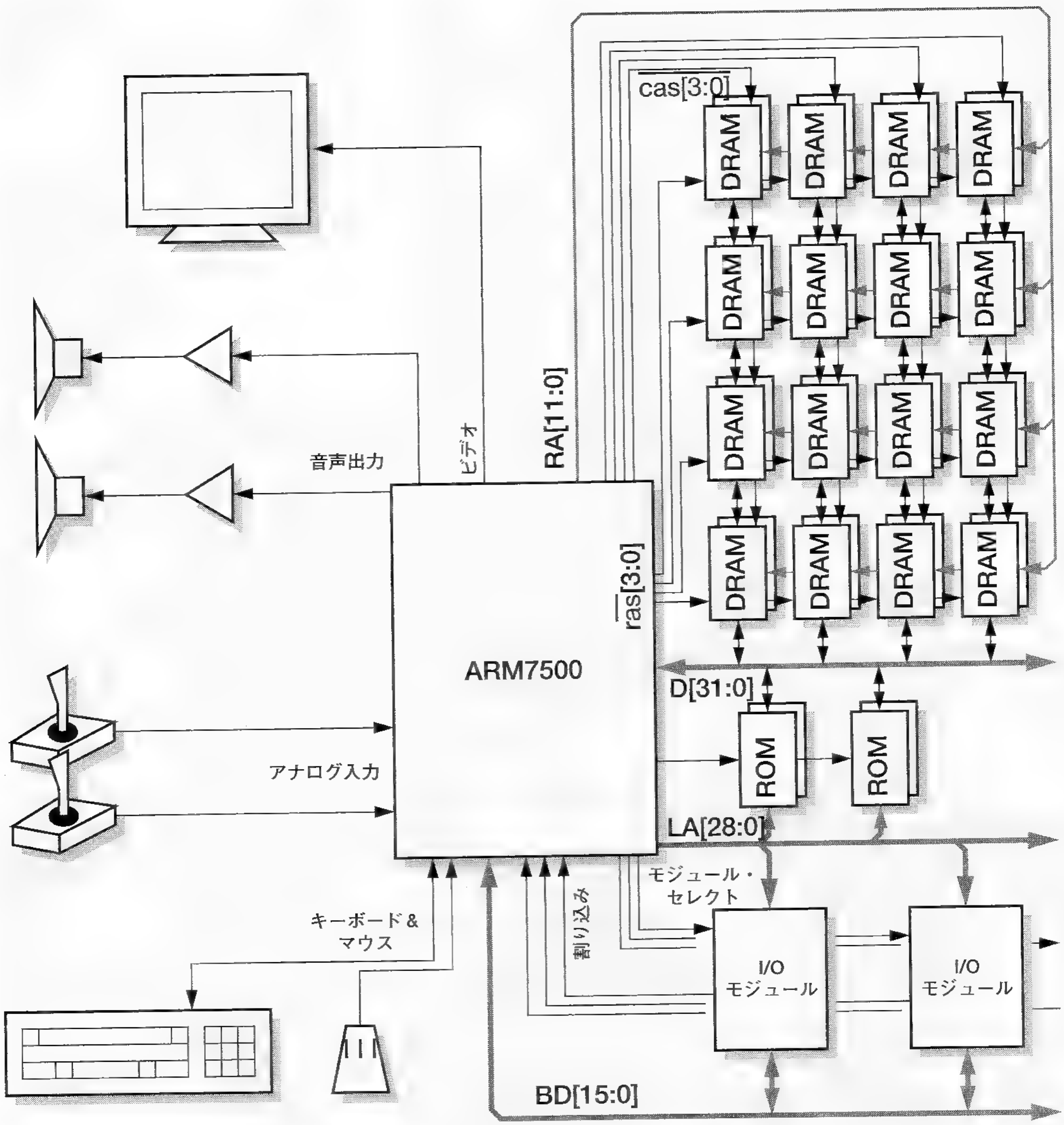
◆メモリおよびI/Oコントローラ

メモリ・コントローラは，最大DRAM4バンクとROM2バンクへの直接接続をサポートします．各バンクは16ビットまたは32ビット幅にプログラムでき，16ビットバンク内の32ビット・アクセスについてはメモリ・コントローラがダブル・アクセスを行います．

DRAMコントローラは，最大256のバースト転送に含まれるシーケンシャル・サイクルについてはページ・モード・アクセスを行い，多様なDRAMリフレッシュ・モードをサポートします．また，ROMコントローラも適切なデバイスを使用すればバースト・モードをサポートします．三つのDMAコントローラがビデオ，カーソル，サウンドの各チャネルを処理します．

I/Oコントローラは，16ビット・オフチップI/Oバス(外部バッファで32ビットに拡張可能)と多数のオンチップ・インターフェースを管理します．オフチップ・バスは，単純な周辺デバイス，インテリジェント周辺モジュール，PCスタイル周辺装置，PCMCIAカード用インターフェースをサポートしています．

オンチップ・インターフェースには，四つのアナログ入力チャネルをサポートするのに使える四つのアナログ・コンパレータがあり，キーボードとマウスの両方もしくは片方に使用する二つのシリアル・ポート，カウンタ・タイマ，8本の汎用オープン・ドレインI/Oライン，プログラマブル割り込みコントローラをサポートします．電力管理機能もあります．



〔図13.10〕ARM7500システムの構成例

◆システム図

チップの使用方法には多数のバリエーションがありますが，典型的なシステム構成を図13.10に示します．

◆アプリケーション

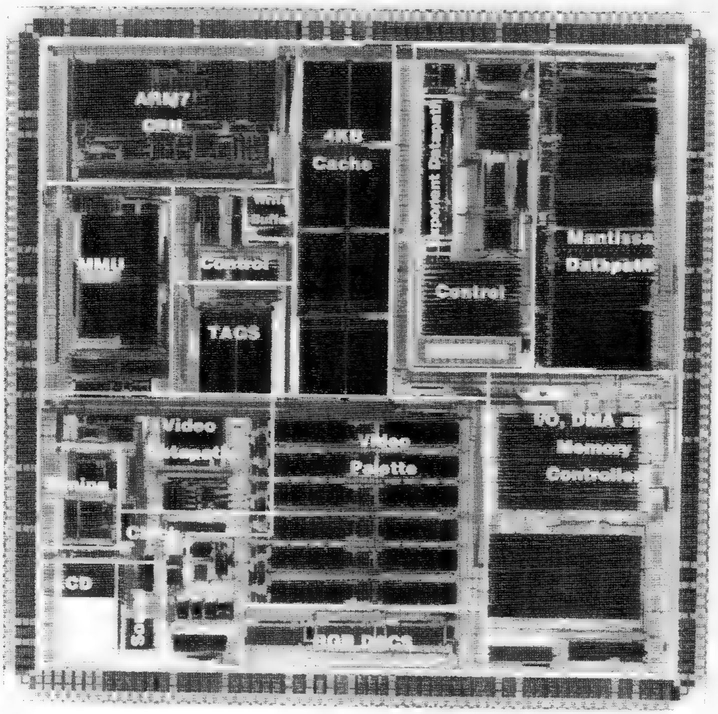
ARM7500は，Acorn Risc PCの低価格版やOnline Mediaインタラクティブ・ビデオ・セットトップボックスに使用されています．もともとのRisc PCチップ・セットと比較した場合のARM7500の主な限界は，標準的なRisc PCの高解像度ディスプレイがVRAMを使用するのに比べ，ビデオ・データ・ストリームが通常の

DRAMに限られることです．このため高解像度ディスプレイでは多数の色を使用することができませんが，液晶ディスプレイやテレビ・モニタ，VGAやスーパVGA解像度では大差ありません．実際は，標準的なDRAMの帯域幅制限により色数が制限されてくるのは，解像度1,280 × 1,024以上になってからです．

インタラクティブ・ビデオおよびゲーム機はテレビ品質のディスプレイ，ポータブル・コンピュータはVGA（640 × 480ピクセル）解像度の液晶ディスプレイを使用しているため，ARM7500はこれらのアプリケーションに理想的と言えます．高集積度と低消費電力という特徴からハンドヘルド・テスト機器に最適ですし，ハイクオリティなサウンドとグラフィックはマルチメディア・アプリケーションに向いています．

◆ARM7500シリコン

図13.11にARM7500FEダイの写真があります．またARM7500の主な特徴は表13.2にまとめました．ダイの左上角にあるARM7コアがダイ領域のたった5%しか占めていないことに注目してください．



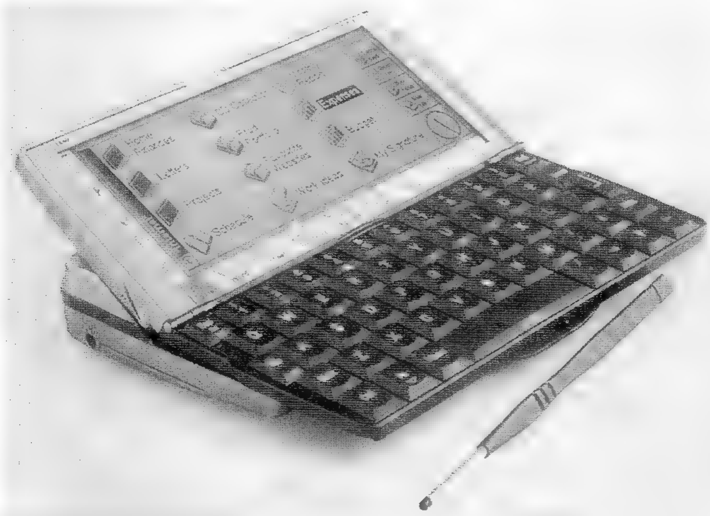
〔図13.11〕ARM7500FEのダイ写真

〔表13.2〕ARM7500の特性

プロセス	0.6 μm	トランジスタ数	550,000	MIPS	30
メタル層	2	ダイ面積	70 mm^2	消費電力	690 mW
V_{dd}	5 V	クロック	0–33 MHz	MIPS/W	43

13.6 ARM7100

ARM7100は高度に集積されたマイクロコントローラで，スマート携帯電話やパームトップ・コンピュータなどさまざまな携帯用アプリケーションに適しています．また，Psionシリーズ5のパームトップ・コンピュータ群で採用されています．図13.12にPsionシリーズ5MX（ARM7100の改訂版を使用）を示します．

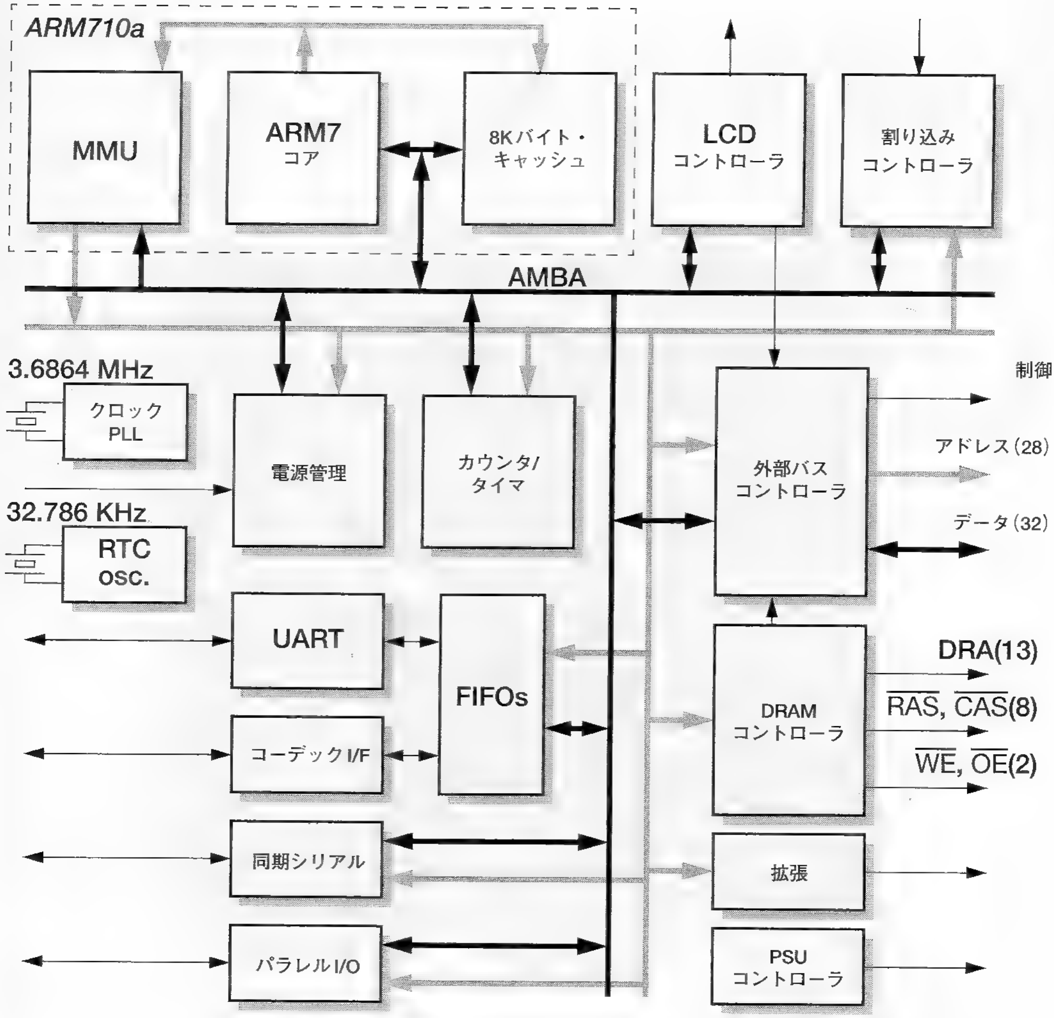


〔図13.12〕Psionシリーズ5MX

◆ARM7100の構成

ARM7100の構成は，図13.13のとおりです．ARM710a CPUはARM7プロセッサ・コア（ARM7TDMIの先行モデルで，Thumbサポートはない），ARMメモリ管理ユニット，8Kバイトの4ウェイ・セット・アソシアティブの4ワード・ライン・キャッシュ，4アドレス8データ・ワードのライト・バッファを組み込んでいます．このクラスの製品でキャッシュ・メモリを使うのは，主にオフチップ・メモリ・アクセスを減らすことによって電力効率を向上するためです．メモリ管理ユニットは，汎用アプリケーションを複数同時に動作するのに必要なオペレーティング・システムをサポートするために必要です．

システム・オンチップ構成は，AMBAバスを中心としています．ペリフェラルには，LCDコントローラ，シリアルおよびパラレルI/Oポート，割り込みコントローラ，オフチップROMに効率的にアクセスするための32ビット外部バス・インターフェース，RAM，DRAMがあります．DRAMコントローラが，このタイプのメモリに必要なマルチプレックス・アドレスおよび制御信号を与えます．



〔図13.13〕ARM7100の構成

◆電力管理

ARM7100は，バッテリ電源をもつ携帯用機器向けに設計されています．このような機器は，ユーザ入力に反応して高性能を発揮する一方，ユーザ入力を待つ間は電力消費レベルを非常に低くする必要があります．このニーズに応えるため，チップには三つの電力管理レベルがあります．

(1) **フル動作モード**．ARM CPUは約14MIPSを出し，3.0Vで24mAを消費します．

(2) **アイドル・モード**．CPUは停止，その他のシステムは動作しており，33mWを消費します．

(3) **スタンバイ・モード**．32kHzクロックだけが動作し，33μWを消費します．

その他，電力効率を向上するための特徴には，ARM7100から何の介入がなくても内容を保存するセルフリフレッシュDRAMメモリのサポートがあります．

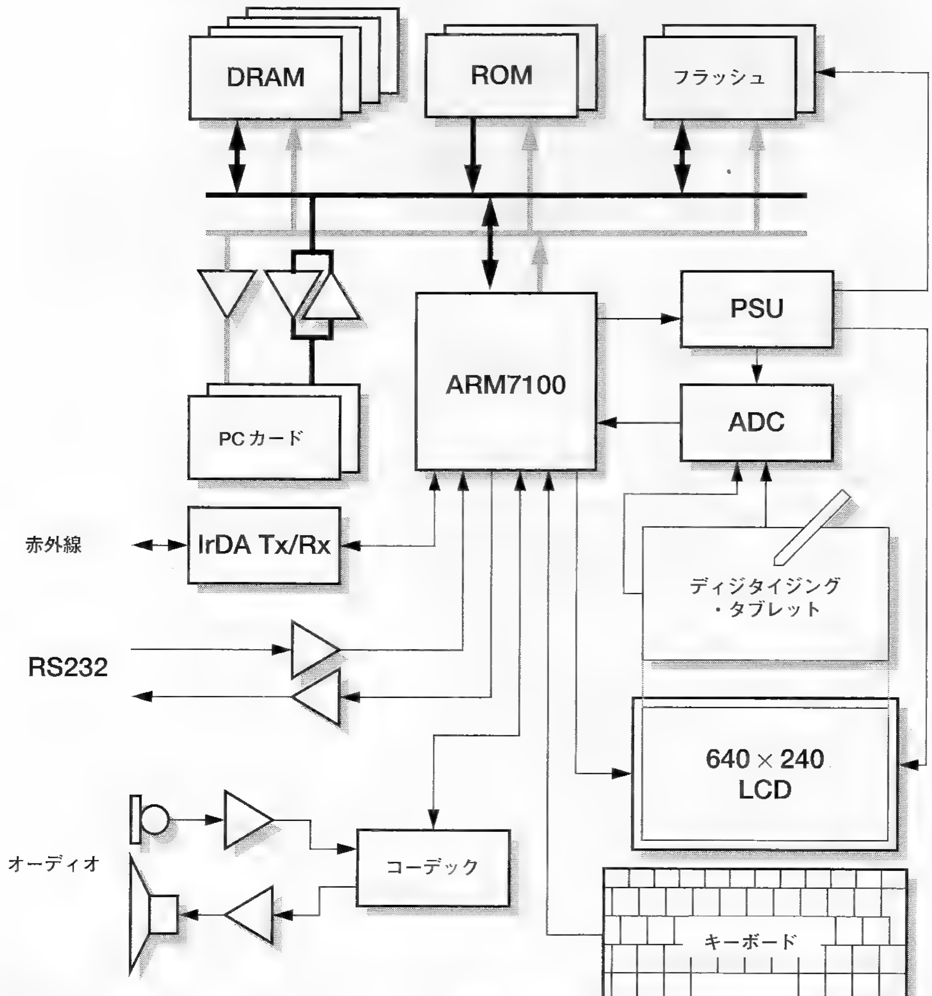
◆Psionシリーズ5

Psionシリーズ5の構成は，図13.14のとおりです．この図は，さまざまなユーザ・インターフェースがどのようにARM7100に接続して，チップ・レベルにおける複雑さを最小限に抑えつつ非常に高度なアーキテクチャをもったシステムを作っているかが示されています．

主なユーザ入力デバイスは，キーボードとスタイラス・ポインタです．前者はパラレルI/Oピンに接続しているだけですが，後者は透明なディジタイジング・タブレットをLCDディスプレイに重ね，アナログ・ディジタル・コンバータ(ADC)経由でインターフェースさせています．

通信デバイスには，有線接続用にRS232シリアル・インターフェースと，プリンタ，モデム，ホストPC(ソフトウェア・ローディングとバックアップ用)への無線接続用にIrDA準拠赤外線インターフェースがあります．オーディオ・コーデックにより音声は，内蔵マイクよりディジタル化してメモリに保存，小型の内蔵ラウドスピーカーを通して後で再生することができます．

ハードウェア拡張は，PCカード・スロットを通して提供します．

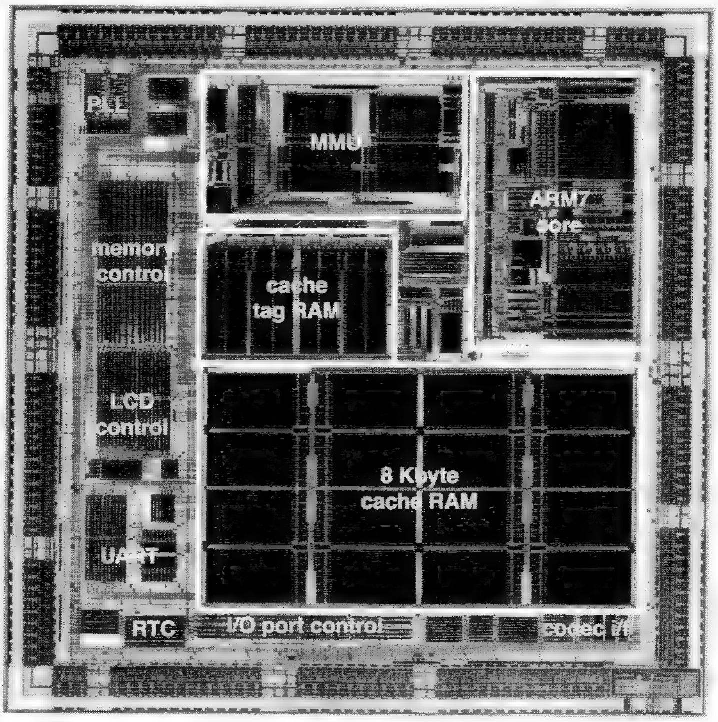


〔図13.14〕Psionシリーズ5のハードウェア構成

◆ARM7100シリコン

ARM7100の主要な機能領域は，図13.15のとおりであり，おもな特徴は表13.3にまとめました．

図13.15を見ると，左端の4分の1のチップ・コア領域を除き，シリコン面積の大半をCPUコアが占めています．8KバイトのキャッシュRAMはCPUコア面積の下半分を占め，キャッシュ・タグ・ストアはその左上にあります．MMUはその上，ARM7コアは右上にあります．ペリフェラル全部とAMBAバスは左端の4分の1にあります．



〔図13.15〕ARM7100のダイ

〔表13.3〕ARM7100の特性

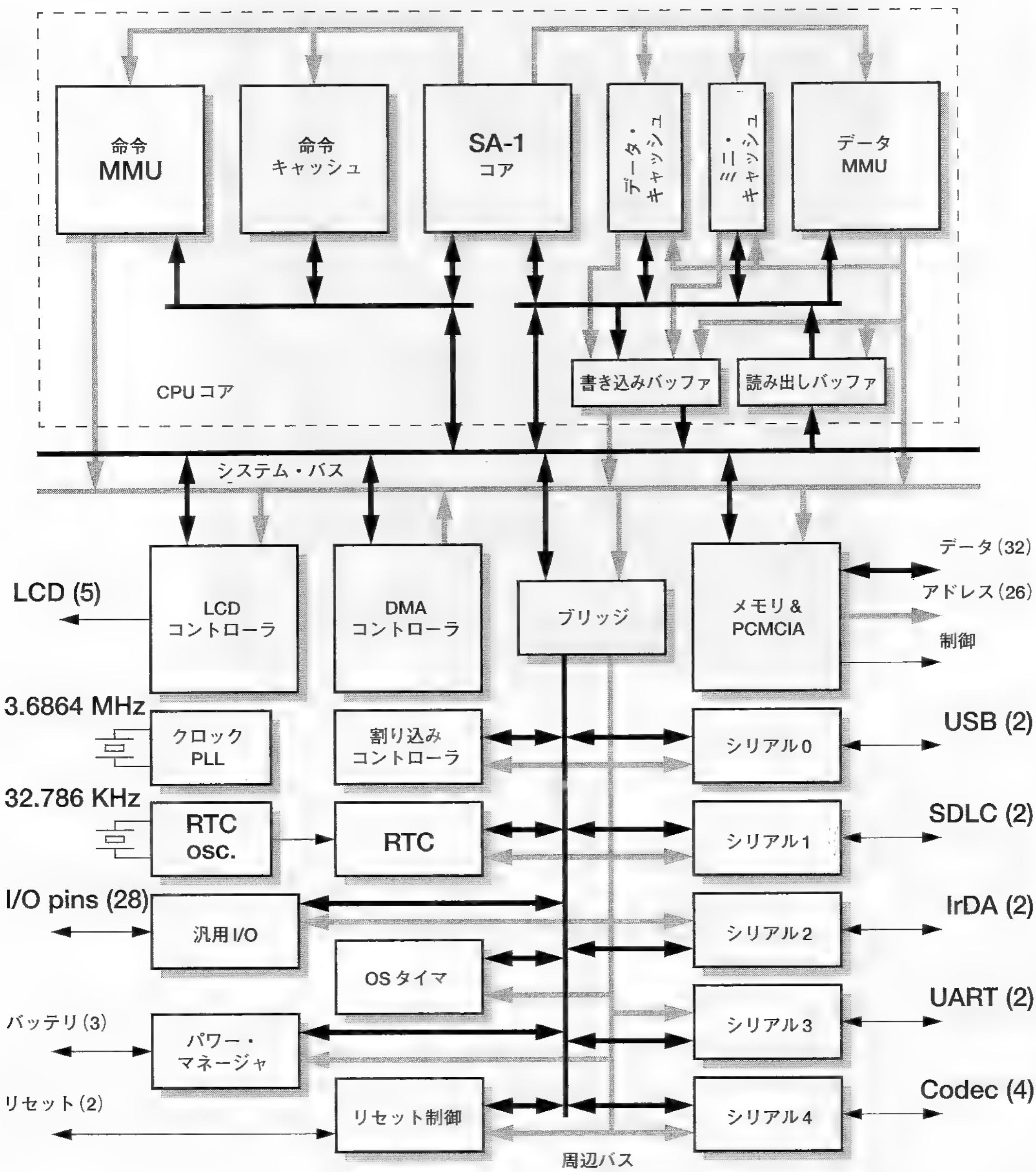
プロセス	0.6 μm	トランジスタ数	N/A	MIPS	30
メタル層	2	ダイ面積	N/A mm^2	消費電力	14 mW
V_{dd}	3.3 V	クロック	18.432 MHz	MIPS/W	212

◆ARM7500FEとの比較

このバランスがARM7500FE（図13.11参照）とは異なっています．ARM7500FE上においてCRTモニタを駆動する高速カラー・ルックアップ・テーブルは，ARM7100におけるLCDコントローラより大きな面積を占めています．ARM7500FE上における浮動小数点ハードウェアも，かなりの面積を占めます．それを埋め合わせるため，ARM7500FEのキャッシュはサイズが半分になっていますが，それでもARM7100より大きな面積を占めます．

13.7 SA-1100

SA-1100は，12.3節で述べたSA-110 StrongARM CPUコアの修正バージョンをベースとした高性能集積システム・オンチップです．SA-1100は，最小限の消費電力で高性能を必要とする携帯電話ハンドセット，モデム，その他のハンドヘルド・アプリケーション向けに設計されています．チップの構成は図13.16のとおりです．



〔図13.16〕SA-1100の構成

◆CPUコア

SA-1100のCPUコアは，SA-110と同じSA-1プロセッサ・コアを使用していますが，WindowsCEに必要な例外ベクタ再配置メカニズムをサポートするよう小さな変更が加えてあります．命令キャッシュも同様で，32ウェイ・アソシアティブCAM-RAM構造で8ワード・ラインを使った16Kバイト・キャッシュです．メモリ管理システムも，WindowsCEに必要なProcessIDメカニズムを組み込んだ以外は変わっていません．

SA-110との大きな違いは，データ・キャッシュ側にあります．もともとの16Kバイト・キャッシュの代わりに512バイトの2ウェイ・セット・アソシアティブ・キャッシュと8Kバイトの32ウェイ・セット・アソシアティブ・キャッシュが並列に組み込まれています．特定のメモリ位置をどちらのデータ・キャッシュに対応させる(対応させるとすれば)かは，メモリ管理テーブルが決定します．この第2の「ミニ・キャッシュ」の目的は，主データ・キャッシュを広範に占有せずに大型のデータ構造をキャッシュすることです．

データ・キャッシュ側におけるもう一つの違いは，読み出しバッファの追加です．このユニットは，プロセッサが要求する前にデータをプリロードし，要求されたときの遅れ時間を短縮します．読み出しバッファは，コプロセッサ・レジスタを通じてソフトウェアで制御します．

CPUコアへの最後の拡張はハードウェア・ブレークポイント・レジスタとウォッチポイント・レジスタの追加で，これもコプロセッサ・レジスタを通じてプログラムします．

◆メモリ・コントローラ

メモリ・コントローラは，32ビット・オフチップDRAM最大4バンクをサポートします．このDRAMは，従来のものでも「拡張データ・アウト(EDO)」でもかまいません．ROM，フラッシュ・メモリ，SRAMもサポートされています．

さらなるメモリ拡張がPCMCIAインターフェース(これには，いくらかの外部「接着」論理が必要)を通じてサポートされています．これは二つのカード・スロットに対応します．

◆システム制御

オンチップ・システム制御機能は以下のとおりです．

(1) **リセット・コントローラ**
(2) **電力管理コントローラ**．ロー・バッテリ警告を制御し，多様なシステム動作モード間を切り替える
(3) **オペレーティング・システム・タイマ・ブロック**．一般的なタイミングおよびウォッチドッグ機能をサポート
(4) **割り込みコントローラ**
(5) **リアルタイム・クロック**．32kHzの水晶発振器から動作
(6) **汎用I/Oピン28本**

◆ペリフェラル

ペリフェラル・サブシステムには，LCDコントローラとUSB，SDLC，IrDA，コーデック，標準UART機能向けの特殊シリアル・ポートがあります．6チャネルDMAコントローラは，CPUを単純なデータ転送動作から開放します．

◆バス構造

図13.16からわかるとおり，SA-1100はブリッジで接続した二つのバスを中心に構築されています．

(1) システム・バスは，すべてのバス・マスタとオフチップ・メモリを接続します．
(2) ペリフェラル・バスは，すべてのスレーブ周辺装置を接続します．

◆アプリケーション

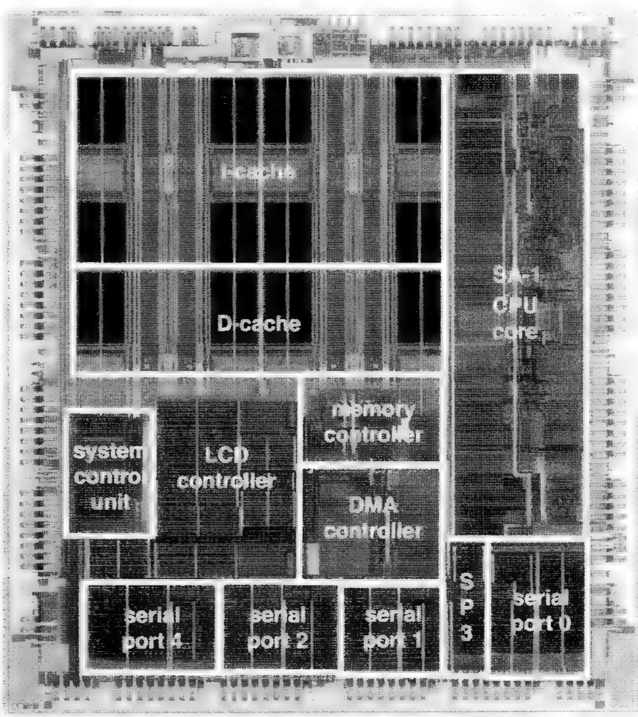
典型的なSA-1100のアプリケーションには，たいていはいくらかのDRAMとROM，またはフラッシュ・メモリ（もしくは両方）を含む一定量のオフチップ・メモリが必要です．あと必要なのは，さまざまなペリフェラル・インターフェース向けのインターフェース回路，ディスプレイなどだけです．結果としてできたシステムはPCBレベルでは非常に単純な一方，処理機能やシステム・アーキテクチャの点では非常に強力で高度なものになります．

◆SA-1100シリコン

SA-1100チップの特性は，表13.4にまとめました．またダイの主要な機能領域は図13.17のとおりです．このチップは，1.5Vという低供給電圧で最適な電力効率を発揮します．また電力効率はいくらか落ちますが，供給電圧を多少高くすると高性能が得られます．

〔表13.4〕SA-1100の特性

プロセス	0.35 μm	トランジスタ数	2,500,000	MIPS	220/250
メタル層	3	ダイ面積	75 mm^2	消費電力	330/550 mW
V_{dd}	1.5/2 V	クロック	190/220 MHz	MIPS/W	665/450



〔図13.17〕SA-1100のダイ写真

13.8 例題と演習

<例題13.1> クリティカルなDSPルーチンを，2ウェイト・ステートのオフチップRAMでなく，ゼロ・ウェイト・ステートのオンチップRAMで実行することによる性能向上を予測しなさい．

典型的なDSPルーチンは積和計算に支配されています．コード・シーケンスの例は次のとおりです．

```
        ..                          ; initialize
LOOP    LDR     r0, [r3], #4        ; get next data value
        LDR     r1, [r4], #4        ; and next coefficient
        MLA     r5, r0, r1, r5      ; accumulate next term
        SUB     r2, r2, #1          ; decrement loop counter
        BNE     LOOP                ; loop if not finished
```

このループには，七つの命令フェッチ(分岐シャドウにある二つを含む)と二つのデータ・フェッチが必要です．乗算にはデータに依存した数の計算サイクルが必要で，標準的なARMコアは，1サイクルに2ビットの値を出します．16ビットの係数の場合，乗算を命令すると係数がサイクル数を決定し，乗算には8内部サイクルが必要になります．さらに，1ロードにつき1内部サイクルかかります．データと係数が常にオンチップ・メモリにある場合，ループはオンチップRAMから実行した場合19クロック・サイクル，2ウェイト・ステートのオフチップRAMから実行した場合さらにウェイト・ステートが14サイクルかかることになります．

したがって，オンチップRAMの使用により，ループは75%スピードアップします．

このスピードアップは，単純にメモリ・アクセス速度を比較した場合に期待される3倍よりはずっと小さいことに注目してください．

演習13.1.1 オンチップRAMを使うことによる電力の節約を予測しなさい(オンチップ・アクセスは2nJ，オフチップ・アクセスは10nJかかるとする)．

演習13.1.2 外部メモリが16ビット幅に限られているとして，再度予測しなさい．

演習13.1.3 プロセッサがThumbモード(プログラムはThumb命令を使って書き直します)と高速乗算をサポートしているとして，32ビットおよび16ビットの各外部メモリについて再度予測しなさい．

<例題13.2> 本章で紹介した設計を調べ，説明されているシステム・チップに採用されている省電力テクニックをまとめなさい．

まず，全システムチップは高度に集積されているため，同じチップ上の2モジュール間でデータを交換する際は常に電力を節約します．

全チップはARMコアをベースにしています．そしてARMコア自体が非常に電力効率的です．

オフチップ・メモリを完全に取り外せるほど十分なオンチップ・メモリをもったチップはありませんが，いくつかのチップは数KバイトのオンチップRAMをもっており，そこにクリティカル・ルーチンをロードして電力を節約できます(そして性能も向上する)．

いくつかのチップはクロック制御回路を組み込んでおり，チップに最大負荷がかかっていない場合はクロッ

ク周波数を落とし（またはクロックを完全に止める），モジュールがアクティブでないときは個別にクロックをスイッチしてオフにします．

全チップはスタティックまたは擬似スタティックCMOS技術に基づいており，少し注意すればディジタル回路がスイッチするときだけ電力を消費するようになります．継続的なバイアス電流を要求するアナログ機能を組み込んだチップもあり，その場合，その回路がアクティブでないときはパワーダウン可能です．

演習13.2.1　組み込み型システムには，通常，クラッシュしたときにシステムを再起動する「リセット」ボタンがありません．ソフトウェア・クラッシュから回復するためにどんなテクニックが用いられていますか？

演習13.2.2　本章で説明したチップのいくつかには，カウンタ・タイマ・モジュールが組み込まれています．それは何のために使用しますか？

演習13.2.3　ARM7500にはオンチップ・キャッシュ，AMULET2e（14.4節参照）にはキャッシュとしても直接アドレスされるメモリとしても構成できるオンチップ・メモリがあります．Ruby IIとVIPにはたんに直接アドレスされるオンチップ・メモリがあります．オンチップRAMとオンチップ・キャッシュの違いを説明し，チップの設計者がなぜ上のような選択をしたのかを説明しなさい．

第14章 AMULET非同期ARMプロセッサ

AMULETプロセッサ・コアは，ARMアーキテクチャの完全な非同期実装です．つまり，セルフタイム式で外部からクロックを供給しなくても動作します．

セルフタイム式ディジタル・システムは，電力効率と電磁環境適合性(EMC)の分野で有利となる可能性を秘めていますが，非同期タイミング・フレームワークを完全に採用するとなるとシステム・アーキテクチャを大幅に設計しなおす必要があり，設計者はそのスタイルで仕事をする新しい心構えをもたなければなりません．セルフタイム設計に対するツール業界のサポートも欠けています．

AMULETプロセッサ・コアは，英国のマンチェスター大学で開発された研究用プロトタイプであり，商業的な未来は不確かです．完全に説明するには，本書はスペースが不足なので，ここでは概略を述べます．これはARM命令セットを実現するまったく異なった方法です．資料については「参考文献」をご覧ください．AMULET1，AMULET2eについて詳しく書かれたものがあるかもしれません．AMULET3についての詳細もいずれ出版されることでしょう．

最新のセルフタイム方式コアであるAMULET3は，Thumb命令セットのサポート，デバッグ・ハードウェア，システムオンチップ・モジュラ設計に対応するオンチップ・マクロセル・バスなど，クロック式のARMコアがもつ特長のほとんどを組み込んでいます．これによって，非同期技術は商業的使用を開始できるレベルに達したわけであり，今後数年もすれば，この技術がARMアーキテクチャの将来に果たす役割があるかどうかが判明することでしょう．

14.1 セルフタイム設計

現在，そしてこれまで四半世紀の間，事実上すべてのディジタル設計が，システムの全コンポーネントの動作を制御するグローバル「クロック」信号を使用してきました．複雑なシステムは複数クロックを使用することもありますが，各クロック・ドメインはやはり同期サブシステムとして設計されています．そして，異なるドメイン間のインターフェースは，信頼性の高い動作を確実にするために極めて慎重に設計する必要があります．

クロックは，つねに支配的であったわけではありません．初期のコンピュータの多くは，部分的な情報を使って部分的な動作を制御する制御回路を採用していました．しかし，そのような「非同期」設計スタイルは，中央クロックを使用した非常に生産的な設計方式が開発され，トランジスタ・リソースの急増をもたらす集積回路技術に対する業界のニーズに応えるようになるとすたれてしまいました．

◆クロック問題

しかし，クロック式の設計スタイルは最近，実用的な問題に直面しています．

(1) クロック周波数が増えるにつれて，グローバルな同期性を維持するのが困難になります．クロック・スキュー(チップ上の異なる点におけるクロック・タイミングの違い)は性能を低下させ，最終的に回路の動作不良を招くこともあります．

(2) またクロック速度が速いと，過度に電力を消費します．クロック分配ネットワークが，チップの総消費電力においてかなりの割合を占めることもあります．クロック・ゲート技術はクロック・ネットの動作に一定の制御力をもつものの，粗い粒度であることが普通であり，クロック・スキューを悪化させることがあります．

(3) グローバルな同期は供給電流の過渡的な変動を大きくし，電磁干渉を起こします．EMC(電磁環境適合性)規制は厳しくなりつつあり，速いクロック速度でその規制に合わせるのは困難になっています．

◆セルフタイム設計への動機

上記の理由で，非同期設計技術は最近また脚光を浴びるようになりました．非同期設計は，上記の問題に以下のように対応します．

(1) クロックがないのでクロック・スキューの問題はありません．

(2) 非同期設計で回路に遷移が起こるのは，要求に応えて有用な動作を実行するときだけです．クロック信号によって継続的に電力を消費することはなく，複雑な電力管理システムによるオーバヘッドもありません．また，ゼロ消費電力と最大性能の間を一瞬で切り替えることができます．組み込み型アプリケーションの多くは作業負荷が急激に変化しますから，非同期プロセッサは大幅に電力を節約する可能性を秘めていると考えられます．

(3) 非同期回路は，内部動作の干渉性が少ないので電磁放射が少なくなります．

そのうえ，同期回路では，最悪時の条件で設計されなければならない一方で，非同期回路では実際の条件で設計するので，最悪時性能でなく実際的な性能を出す可能性があります．

AMULETシリーズの非同期マイクロプロセッサは，非同期設計への世界的な研究の高まりの一環として英国のマンチェスター大学で開発されました．他の場所で開発された非同期マイクロプロセッサの例もありますが，他の命令セットアーキテクチャをサポートしており，ARM命令セットを実装しているのはAMULETプロセッサだけです．

◆セルフタイム・シグナリング

非同期設計は，さまざまな側面とアプローチをもつ複雑で困難な作業です．非同期設計を全体的に紹介するのは本書の目的から外れてしまいますが，基本的な概念をいくつか紹介すれば，読者はAMULETコアのもっとも重要な特徴を把握することができるでしょう．これらの概念の中でもっとも重要なのは，非同期通信という発想です．参照クロックなしで，どうやってデータ・フローを制御するのでしょうか？

AMULET設計は両方とも，要求-応答確認ハンドシェイクを使ってデータ・フローを制御します．送信側から受信側へのデータ通信を構成する動作シーケンスは以下のとおりです．

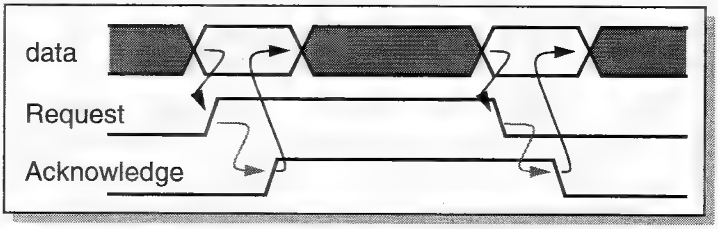
1. 送信側が有効データ値をバス上に置く
2. 送信側が要求イベントを発行する
3. 受信側が準備できたらデータを受け入れる

4. 受信側が送信側に応答確認を発行する
5. 送信者はデータをバスから取り除き，準備できたら次の通信を開始する

データは，従来のバイナリ・エンコーディングを使ってバス上を受け渡しますが，要求イベントと応答確認イベントを信号として送出するには二つの方法があります．

◆遷移信号通信プロトコル

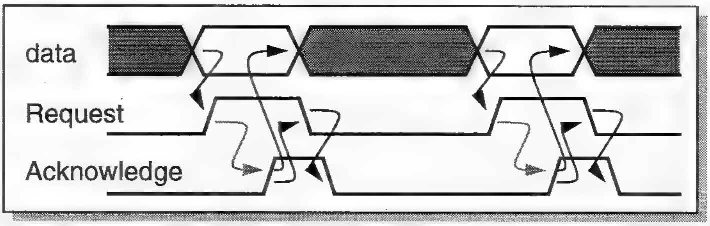
AMULET1は，レベルにおける変化（“L”から“H”，“H”から“L”のいずれか）がイベントを知らせるという，遷移エンコーディングを使用します．これは，図14.1に示したとおりです．



〔図14.1〕遷移信号通信プロトコル

◆レベル信号通信プロトコル

AMULET2とAMULET3は，立ち上がりエッジがイベントを知らせ，次のイベントを知らせる前にゼロ復帰フェーズが必要なレベル・エンコーディングを使用します．これは，図14.2に示したとおりです．



〔図14.2〕レベル信号通信プロトコル

遷移信号通信プロトコルは概念的に明確なので，AMULET1で使用されました．遷移はすべて役割を担っているため，そのタイミングは回路の機能によって決まります．また，遷移数が最小限なので低消費電力です．しかし，遷移制御を実装するのに使っているCMOS回路は比較的低速で非効率的なので，AMULET2とAMULET3ではレベル信号通信プロトコルを使用しています．これは，遷移数が2倍になるにもかかわらず，もっと高速で低消費電力の回路を使用します．ただし，プロトコルにおける復帰（ゼロ復帰）フェーズのタイミングについては，任意の決定が下される余地があります．

◆セルフタイム・パイプライン

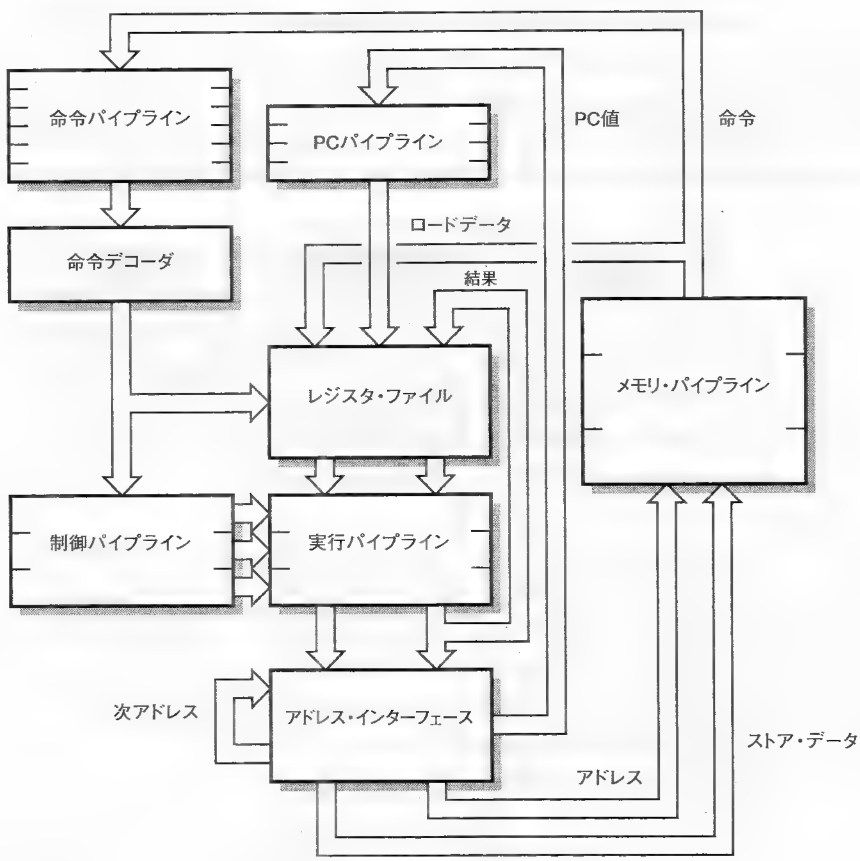
セルフタイミング・テクニックを使って非同期パイプライン処理ユニットを構築すると，各ステージに処理遅延を設け，上記プロトコルのいずれかによって結果を次のステージに送ることができます．回路が正しく設計されていれば，多様な処理および外部遅延に対応することができます．重要なのは，イベントのローカル・

シーケンスだけです（ただし，もちろん長い遅延は性能低下につながる）．

同期パイプラインでは，全パイプラインが最悪時の環境条件（電圧と温度）におけるもっとも遅いステージのクロック速度に合わされ，最悪時のデータを予測していますが，非同期パイプラインは現在の条件によって決まる可変速度で動作します．まれに起こる最悪時条件によって処理ユニットに時間がかかることはあります．万一このような条件が起こったときに性能が低下することもあります．しかし，その頻度が少ないので性能全体に与える影響は小さくなります．

14.2 AMULET1

AMULET1プロセッサ・コアは，図14.3のように高レベルの構成をもっています．この設計には，それぞれのタイミング，それぞれの速度で動作しつつ相互動作する一連の非同期パイプラインが組み込まれています．このようなパイプラインは，とんでもない遅延時間をプロセッサにもたらすように思えるかもしれませんが，同期パイプラインと違って非同期パイプラインの遅延時間は非常に短くなる可能性があります．



〔図14.3〕AMULET1の内部構成

プロセッサの動作は，アドレス・インターフェースが命令フェッチ要求をメモリに発行することから始まります．アドレス・インターフェースには独立したアドレス・インクリメンタがあり，これによって多様なパイプライン・バッファの容量が許す限り多くの命令をプリフェッチすることができます．

◆アドレス非決定性

データ転送命令や予測不能な分岐など，新しいメモリ・アドレスを生成する必要のある命令は，実行パイプラインで新しいアドレスを計算し，それをアドレス・インターフェースに送ります．このアドレスは，アドレス・インターフェース内の内部インクリメント・ループに対して任意のタイミングで到着しますから，新しいアドレスをアドレス・ストリームに挿入するポイントは非決定的です．したがってプロセッサが分岐命令を超えてプリフェッチする深さは根本的に非決定的なのです．

◆レジスタ整合性

実行パイプラインが効率的に動作するためには，レジスタ・ファイルは，前の命令の結果が戻る前に命令のオペランドを発行する必要があります．しかし，オペランドが前の結果に依存している場合もあります(これがリード・アフターライト・ハザード)．この場合，パイプライン中でさらに先にある正しい値を供給するフォワーディング・メカニズムがない限り，レジスタ・ファイルは結果が戻るまでオペランドを発行できません．

多くのRISCプロセッサ(ARM8，StrongARMなど)に使用されているレジスタ・フォワーディング・メカニズムは，同期パイプラインの性質に基づいています．なぜなら一つのパイプライン・ステージにあるソース・オペランド・レジスタ番号と，別のステージにあるデスティネーション・レジスタ番号を比較するからです．非同期パイプラインでは，各ステージが全部異なるタイミングで動いていますから，比較するステージ間を明示的に同期して非同期動作の利点をほとんどなくしてしまわない限り，そのような比較はできません．

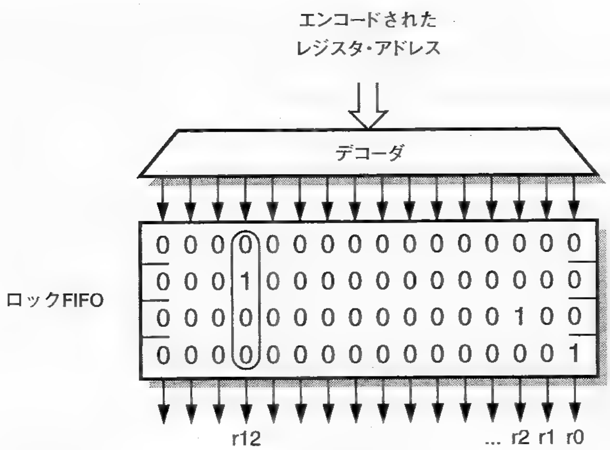
◆レジスタ・ロッキング

AMULET1では，レジスタ・ロックFIFOに基づく新しい形のレジスタ・ロッキングを使ってレジスタの整合性を保持しています．デスティネーション・レジスタ番号は，関連する結果が実行またはメモリ・パイプラインからレジスタ・バンクに戻されるまで，デコードされた形でFIFO内にストアされます．

ロックFIFOの構成は，図14.4のとおりです．FIFOの各ステージは，デスティネーション・レジスタに対応する位置に「1」をもっています．この図では，FIFOが16レジスタを制御しており(AMULET1の場合，FIFOは，ARMのバンク・レジスタを含め各物理レジスタにつき1列をもっている)，到着する最初の結果がr0に，2番目がr2に，3番目がr12に書き込まれ，4番目のFIFOステージが空であるような状態が示されています．

次の命令がソース・オペランドとしてr12を要求すると，r12に対応する(図の囲み内)FIFO列を検査するとr12が有効かどうかがわかります．列のどこかに「1」があると保留中のr12への書き込みがあることを示しますから，現在の値は旧いことになります．読み出しは「1」がクリアされるまで待ってから進行します．

この「検査」は各レジスタの列を横切る論理「OR」関数によってハードウェアで実装されています．FIFO内のデータは「OR」出力が使用されている間にFIFO内を移動するかもしれませんから，これは危険に思えるかもしれません．しかしデータは，一つのステージから別のステージへ複製されることによって非同期FIFO内を移動します．2番目のステージに移って初めて最初のステージからデータが消去されますから，伝播する「1」は一つまたは二つの位置に交代で出現することになり，まったく消えてしまうことはありません．したがって「OR」出力はデータが移動中であっても安定しています．



〔図14.4〕AMULETレジスタ・ロックFIFO構成

◆AMULET1の性能

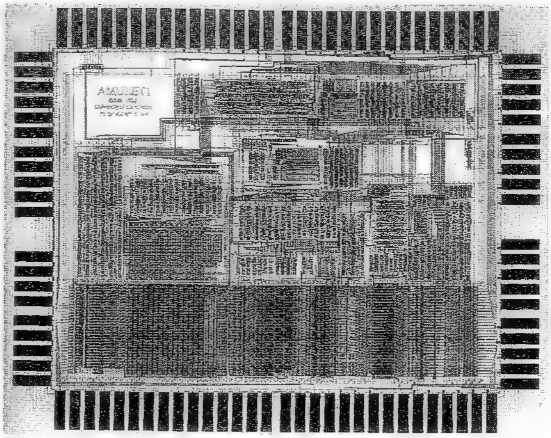
AMULET1は，商用マイクロプロセッサ・アーキテクチャの完全な非同期実装の設計が現実に可能であることを証明するために開発されました．プロトタイプ・チップは動作し，標準的なARM開発ツールを使って生成したテスト・プログラムを実行しました．プロトタイプの性能は，**表14.1**にまとめたとおりです．この表には，European Silicon Systems (ES2) とGEC Plessey Semiconductors (GPS) による2種類のプロセス・テクノロジで製造されたデバイスの特徴を示しています．1μm AMULET1コアのレイアウトは，図14.5のとおりです．Dhrystoneベンチマークに基づく性能値は，同じプロセス・テクノロジで製造されたARM6プロセッサと同じレベルであり，決してそれより高くはありません．しかし，AMULET1は主にセルフタイム設計の実行可能性を証明する目的で製造されたものであり，その目的は明らかに果たしています．

〔表14.1〕AMULET1の特徴

	AMULET1/ES2	AMULET1/GPS	ARM6
プロセス	1 µm	0.7 µm	1 µm
サイズ(mm²)	5.5 x 4.1	3.9 x 2.9	4.1 x 2.7
トランジスタ数	58,374	58,374	33,494
性能	20.5 kDhrystones	~40 kDhrystones[a]	31 kDhrystones
乗算器	5.3 ns/bit	3 ns/bit	25 ns/bit
動作条件	5 V, 20°C	5 V, 20°C	5 V, 20 MHz
消費電力	152 mW	N/A[b]	148 mW
MIPS/W	77	N/A	120

a. 最大値見積もり

b. GPS部は電力消費を測定できない



〔図14.5〕AMULET1ダイ・プロット

14.3 AMULET2

AMULET2は，第2世代の非同期ARMプロセッサです．構造は，図14.3に示したAMULET1のものと非常によく似ています．前に述べたように，AMULET1で使っていた2相（遷移）シグナリングは廃止し，4相（レベル）シグナリングを使っています．さらに性能を向上させるために，多数の構造的特長が追加されました．

◆AMULET2レジスタ・フォワーディング

AMULET2は，AMULET1と同じレジスタ・ロッキング・メカニズムを使用していますが，レジスタ依存性によるストールで性能が低下するのを防ぐため，やはりフォワーディング・メカニズムを組み込んでひんぱんに起こるケースを処理しています．同期プロセッサ・パイプラインに使用されるバイパス・メカニズムは非同期パイプラインには適用できませんから，新しいテクニックが必要です．AMULET2では，以下の二つのテクニックを使用しています．

(1) 最終結果レジスタ． 命令デコーダは，実行パイプラインから得た結果のデスティネーションの記録をとり，直後の命令がこのレジスタをソース・オペランドとして使用していれば，レジスタの読み出しフェーズをバイパスして最終結果レジスタから値を取っていきます．

(2) 最後にロードしたデータ・レジスタ． 命令デコーダは，メモリからロードした最終データ・アイテムのデスティネーションの記録をとり，このレジスタがソース・オペランドとして使用されると常に，読み出しフェーズをバイパスして最後にロードしたデータ・レジスタから直接値を取り出します．ロックFIFOに似たメカニズムがレジスタのガードとして働き，必ず正しい値を取り出せるようになっています．

上記二つのメカニズムは，必要な結果が入手可能であることに依存しています．多少でも不確かである場合（結果が条件実行した命令によって生成されている場合など），命令デコーダはロッキング・メカニズムを使って，オペランドの供給における多様な遅延に対応する非同期構成の能力を生かすことができます．

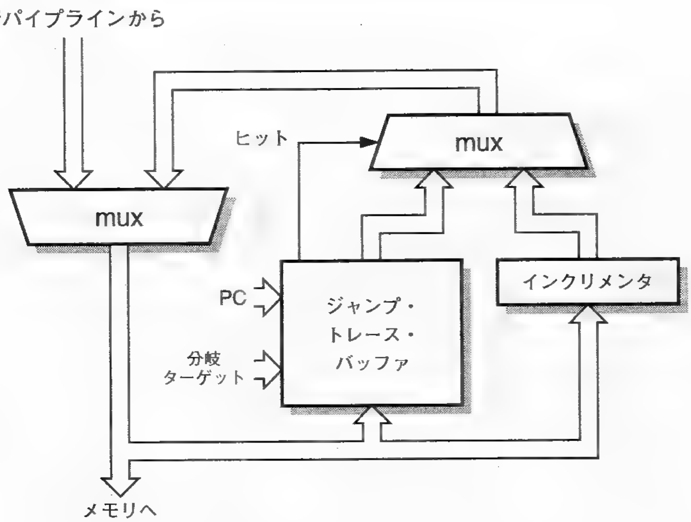
◆AMULET2ジャンプ・トレース・バッファ

AMULET1は，現在のPC値からシーケンシャルに命令をプリフェッチし，シーケンシャル実行と異なるも

のはすべて実行パイプラインからアドレス・インターフェースへの修正として発行する必要があります．PCを修正するたびに性能が低下し，捨てた命令をプリフェッチするためにエネルギを消費します．

AMULET2では，前に分岐がいつ成立したかを記憶し，そのあと同じパスを制御がたどることを推測することによって，この非効率性を下げようとしています．このジャンプ・トレース・バッファの構成は，図14.6のとおりです．1969年から1974年の間にマンチェスター大学で開発されたMU5メインフレーム・コンピュータ（これも非同期制御で動作）で使用されていたものと似ています．

このバッファは，プログラム・カウンタと最近成立した分岐命令のターゲットをキャッシュし，すでにストアされたアドレスからの命令フェッチを検出するたびに予測された制御フローをシーケンシャルから前の分岐ターゲットに変更します．この予測が正しいことが判明すると，正しい命令シーケンスがメモリからフェッチされます．予測が間違っていて分岐が成立すべきでなかった場合は，分岐は「非分岐」命令として実行され，前のシーケンシャル・フローに戻ります．



〔図14.6〕AMULET2ジャンプ・トレース・バッファ

◆分岐統計

ジャンプ・トレース・バッファの有効性は，典型的な分岐動作についての統計によって決まります．典型的な数値は表14.2のとおりです．

〔表14.2〕AMULET2分岐予測統計

予測アルゴリズム	正解	不正解	冗長フェッチ
シーケンシャル	33%	67%	2/分岐 （平均）
トレース・バッファ	67%	33%	1/分岐 （平均）

ジャンプ・トレース・バッファがない場合，デフォルトのシーケンシャル・フェッチ・パターンでは全分岐が成立しなかったと予測しているのと同じです．これは全分岐の3分の1については正しく，残りの3分の2につ

いては間違っています．約20エントリをもつジャンプ・トレース・バッファを使うと，この割合が逆転し，全分岐の約3分の2を正しく予測し，約3分の1については予測を間違うか予測に失敗します．

分岐を超えたプリフェッチの深さは非決定的ですが，約3命令のプリフェッチ深さはAMULET2上で観察できます．フェッチした命令は，分岐が正しく予測された場合には使用しますが，分岐が間違って予測されたり予測されなかったりした場合は廃棄します．したがってジャンプ・トレース・バッファは，重複するフェッチを1分岐当たり2から1に減らします．典型的なコードでは，およそ5命令に一つが分岐命令ですから，ジャンプ・トレース・バッファを使うと命令フェッチ帯域幅を約20％低減，総メモリ帯域幅を10-15％低減すると期待できます．

使用可能なメモリ帯域幅によってシステム性能に制限がある場合，この節約は直接，性能向上につながります．また，重複動作をなくすことによって消費電力を低減します．

◆"Halt"命令

他のマイクロプロセッサと違って，ARMは明示的な「Halt」命令をもっていません．代わりに，有用な作業がなくなるとプログラムは通常アイドル・ループを開始し，以下を実行します．

```
        B     .       ; loop until interrupted
```

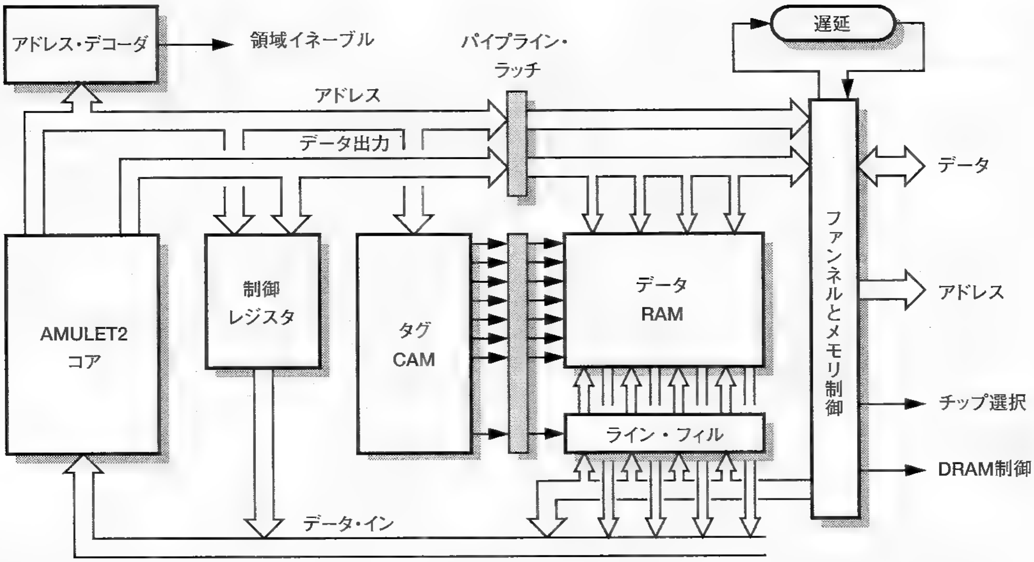
ここで「.」は現在のPCを示しますから，分岐ターゲットは分岐命令そのものとなり，割り込みによって他のことをするまでプログラムはこの単一命令ループを継続します．

プロセッサはこのアイドル・ループの間，何も有用な作業はしていませんから，使われる電力は無駄になります．AMULET2は，自分自身へループする分岐に対応するオペコードを検出し，それを使って非同期制御ネットワークの1点で信号をストールします．このストールは制御全体に伝播し，プロセッサをゼロ電力の非アクティブな状態にします．アクティブな割り込み要求があるとストールは解除され，プロセッサはすぐに通常のスループットを再開することができます．

したがってAMULET2は，ゼロ電力状態と最大スループット状態を非常に速い速度で切り替えることができ，ソフトウェア・オーバヘッドはありません．実際，ARMコードの大部分は，たとえこの方法を意識して書かれていなくても，この方法で最適な電力効率を発揮しています．そのため，バースト的なリアルタイム負荷特性をもった低電力アプリケーションに最適なプロセッサとなっています．

14.4 AMULET2e

AMULET2eはAMULET2のプロセッサ・コア（14.3節を参照）で，キャッシュまたは固定RAM領域として構成可能な4Kバイト・メモリ，そしてDRAMから作ったメモリを含む8，16，32ビットの外部デバイスを直接接続できる柔軟なメモリ・インターフェース（いわゆる「funnel」）を組み合わせています．AMULET2eの内部ブロック図を図14.7に示します．



〔図14.7〕AMULET2e内部構成

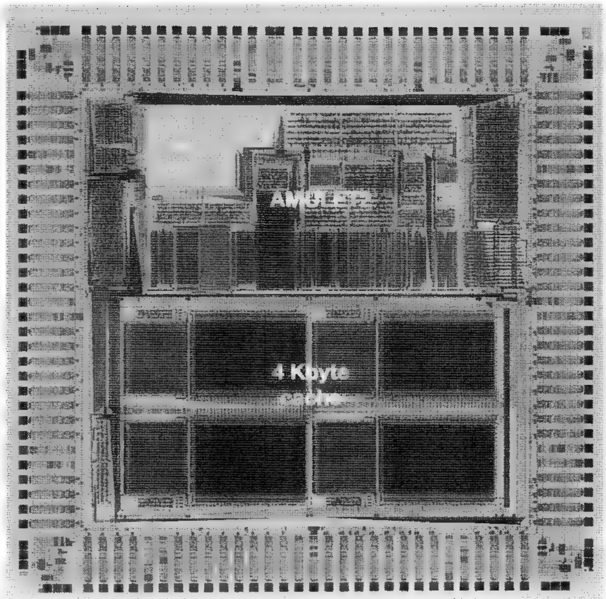
◆AMULET2eのキャッシュ

キャッシュは四つの1Kバイト・ブロックで構成されています．各ブロックはフルアソシアティブ・ランダム置換ストアで4ワードのラインおよびブロック・サイズをもっています．CAMとRAMセクション間のパイプライン・レジスタにより，RAM内で前のアクセスが完了している間に，次のアクセスがCAMルックアップを開始できます．つまり，最初のメモリ要求のデータが戻ってくる前に複数メモリ要求を発行できるというAMULET2コアの能力を生かしているのです．シーケンシャル・アクセスを検知するとCAMルックアップをバイパスし，電力を節約すると同時に性能を向上します．

キャッシュ・ライン・フェッチはブロックされませんから，アドレスした項目にまずアクセスして，そのあとラインの残りがフェッチされている間，プロセッサは動作を継続できます．ライン・フェッチ・オートマトンはプロセッサがキャッシュにアクセスする間，ライン・フェッチ・バッファをロードし続けます．追加されたCAMエントリが，ライン・フェッチ・バッファにストアされたデータへの参照を識別します．たしかに，データはライン・フェッチ・バッファに残っているため，キャッシュのデータと同様にアクセス可能です．ただし，次にキャッシュがミスになると，バッファはすべてキャッシュにコピーされ，新しいデータが外部メモリからライン・フェッチ・バッファにロードされます．

◆AMULET2eシリコン

AMULET2eダイの主要な機能領域は図14.8のとおりであり，デバイスの特徴は表14.3にまとめてあります．AMULET2コアは93,000個，キャッシュは328,000個のトランジスタを使用し，残りは制御論理とパッドになります．



〔図14.8〕AMULET2eダイ・プロット

〔表14.3〕AMULET2eの特性

プロセス	0.5 μm	トランジスタ数	454,000	MIPS	40
金属レイヤ	3	ダイ面積	41 mm^2	消費電力	140 mW
V_{dd}	3.3 V	クロック	none	MIPS/W	285

◆タイミング・リファレンス

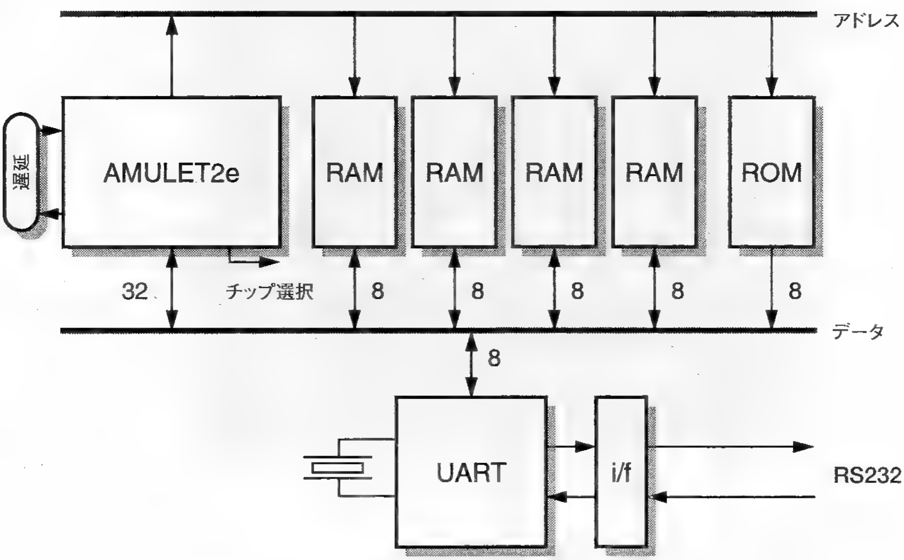
非同期システムにリファレンス・クロックがないため，メモリ・アクセスのタイミングには注意が必要です．このためAMULET2eでは，チップに直接接続した単一外部リファレンス遅延と，起動時にロードするコンフィグレーション・レジスタにより，各メモリ領域の構成とタイミング・プロパティを指定します．リファレンス遅延は通常，外部SRAMアクセス時間を反映するため，RAMは1リファレンス遅延をとるよう設定します．遅いROMは数リファレンス遅延をとるよう設定します（リファレンス遅延は，オフチップ・タイミングにのみ使用されることに注意のこと．オンチップ遅延はすべてセルフタイムである）．

◆AMULET2eのシステム

AMULET2eは，小型システムの構築をなるべく簡単にするよう構成されています．たとえば，図14.9にAMULET2eを組み込んだPIEカードが図示してあります．AMULET2eそのもの以外には，SRAMチップ四つ，ROMチップ一つ，UART，RS232ライン・インターフェースしかありません．UARTは水晶発振器を用いてビットレートを制御しますが，システム・タイミング機能についてはすべてAMULET2eが制御します．

ROMは，標準のARM「Angel」コードをもち，ホスト・コンピュータはRS232シリアル・ラインのもう片方

の端で標準的ARM開発ツールを動作します．このシステムは，メモリ・インターフェースさえきちんと配慮されていれば，非同期型プロセッサの使用が従来のクロックされたプロセッサの使用より，決して難しくないことを証明しています．



〔図14.9〕AMULET2eのPIEカード構成

14.5 AMULET3

AMULET3は，非同期設計の商用的な実現可能性を確立するために開発中です．先行モデルと同様，AMULET3もフル機能を備えたARM互換マイクロプロセッサで，割り込みとメモリ・フォールトをサポートしています．AMULET1とAMULET2はARM6アーキテクチャ（ARMアーキテクチャ・バージョン3G）を実装していました．AMULET3は，16ビットThumb命令セットを含むARMアーキテクチャversion4Tをサポートしています．

◆目標性能

AMULET3プロジェクトの目標は，ARM9TDMIと電力効率と性能で匹敵するようなARMアーキテクチャ4Tの非同期実装を製作することでした．つまり，性能ターゲットは0.35 μmプロセス上で100MIPS以上（Dhrystone 2.1で測定）です．AMULET2eでは，0.5 μmプロセス上で40MIPSでした．

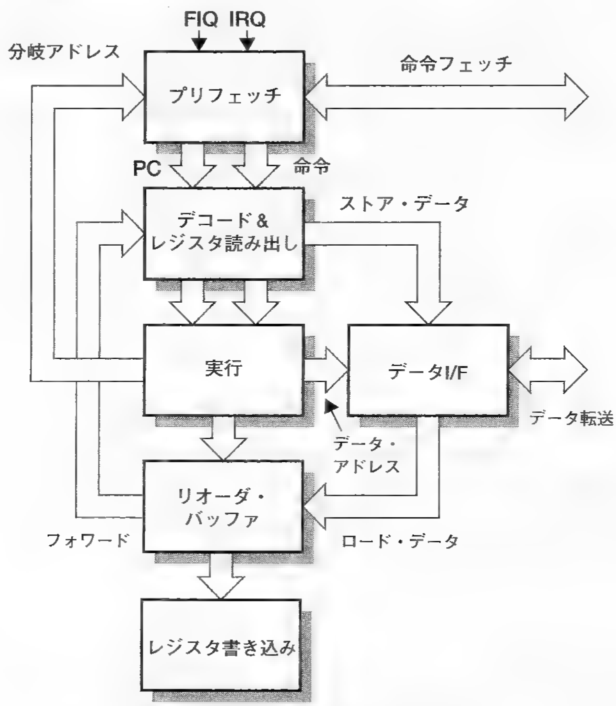
性能を2倍以上に向上するには，コア構成を革新的に変更する必要があります．同期プロセッサでは，プロセッサ・パイプラインのサイクル速度を高めながら命令当たりの平均サイクル数を少なくするのが基本的なアプローチです．しかし，非同期設計の「サイクル」は外部クロックによって決まるものではなく，周期は決まっていません．

◆AMULET3コアの構成

AMULET3の構成は，図14.10のとおりです．次の6段のパイプライン・ステージをもちます．

- 命令プリフェッチ・ユニット．分岐ターゲット・バッファを含む
- 命令デコード．レジスタ読み出しおよびフォワーディング・ステージ
- 実行ステージ．シフタ，乗算器，ALUを含む
- データ・メモリ・インターフェース
- リオーダ・バッファ
- レジスタ結果ライトバック・ステージ

このコアは，高性能を発揮するのに必要な高いメモリ帯域幅を提供するハーバード・アーキテクチャ(命令メモリ・ポートとデータ・メモリ・ポートが別々)を採用しています．データ・メモリ・アクセス(ロードとストア)を必要とする命令だけがデータ・メモリを通過しますから，このような命令については単純なデータ処理命令よりパイプラインが深くなります．



〔図14.10〕AMULET3コア構成

◆命令プリフェッチ・ユニット

プリフェッチ・ユニットは自律的に動作し，ストアする余地があればいつでも32ビット命令(ARM命令一つまたはThumb命令二つ)パケットをメモリからフェッチします．

プリフェッチ・ユニットは，AMULET2で使用しているジャンプ・トレース・バッファと似た分岐予測ユニットを組み込んでいますが，Thumbコードでの分岐予測をサポートする拡張を備えています．2命令のThumbパケットは，分岐命令を0～2の範囲で含むことがあり，分岐予測器はこれらの場合すべてを処理しな

ければなりません．Thumbコードを実行する場合，16エントリ分岐予測ユニットは，8エントリずつの2部分に分かれて動作し，偶数のハーフ・ワード・アドレスにおける分岐を片方にストアし，奇数のハーフ・ワード・アドレスにおける分岐をもう片方にストアします．各ハーフ・ワードに分岐をもつパケットは両方の部分で「ヒット」する場合がありますが，この場合は最初の命令のターゲット（偶数アドレスにおけるもの）が優先されます．

AMULET3はAMULET2と同様，ゼロ電力「停止」命令を含んでいます．AMULET3では，停止は実行ユニットでなくプリフェッチ・ユニットで実行するため，割り込み発生時に早く最大性能に回復します．割り込み自体もここで処理されますから，これも待ち時間を低減します．

◆デコードおよびレジスタ読み出し

デコード・ステージはThumbおよびARMデコード論理のほか，レジスタ読み出しメカニズム，フォワーディング・メカニズムを含みます．

ARM7TDMIは，まずThumb命令を対応するARM命令に変換し，ARMデコードを実行することによってThumbデコード機能を実現していました．またARM9TDMIは，Thumb命令を直接デコードすることによってデコード待ち時間を低減しました．AMULET3はその中間の方法を採用し，一部のタイム・クリティカルな制御信号を直接実行，その他はARM命令デコーダを経由します．非同期パイプラインでは動作に多少の伸縮性があり，ARMコードの動作中は効果的に折りたたまれる独立したThumbデコード・ステージを備えています．

レジスタ読み出し/フォワーディング・ステージは，レジスタ・ファイルからオペランドを回収し，リオーダ・バッファの中でさらに新しい値を検索することによって正しい最新値を実行ユニットに供給します．正しい値がまだ入手できない場合は，入手できるまでフォワーディング・プロセスはストールします．

StrongARM設計者のアイデアを引き継ぎ，AMULET3レジスタ・ファイルには三つの読み出しポートがあります．これによりほとんど全部の命令が1サイクルで発行できます．

◆実行

実行ステージは，加算器，シフタ，乗算器を含み，PSRも論理的にはこのステージに入ります．乗算器は1サイクルに8ビットの積を算出しますが，通常のプロセッサ・サイクル時間よりずっと速いサイクルなので，約20nsecで32×32積を算出します．加算器は，ARM9TDMIで使っているのと同じキャリ・アービトレーション方式を使用します（「キャリ・アービトレーション加算器」参照）．

◆データ・メモリ・インターフェース

データ・メモリ・インターフェースは，ロード命令とストア命令で別のパイプライン・ステージとして動作しますが，その他の命令の場合はバイパスされます．したがって遅いメモリ位置をアドレスするロード命令が，パイプライン内にある他の命令を（それらがロードした値を使用しない限り）止めることがありません．

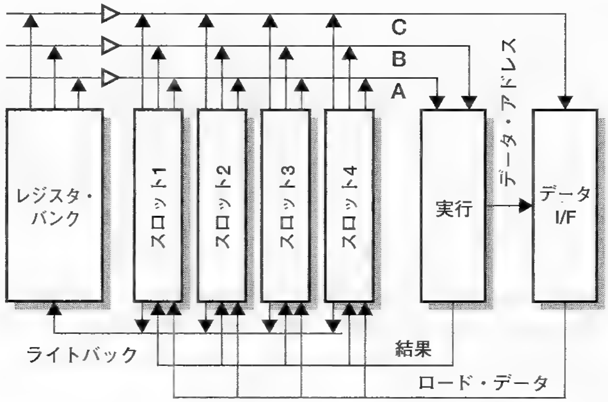
◆リオーダ・バッファ

リオーダ・バッファは，実行ユニットとデータ・メモリ・インターフェースからの結果を入手可能になった順番で受け入れ，プログラム順にレジスタ・ファイルにライトバックできるようになるまでストアします．いったん結果がリオーダ・バッファに入ると，その結果を必要とする命令にフォワーディングすることができます．

この機能を果たすのに必要な構造は，図14.11のとおりです．バッファには四つの「スロット」があり，命令

が発行されると命令が生成した各結果に1スロットが割り当てられます(スロットは，メモリ・フォールトが正しく処理されるよう命令を「ストア」するためにも割り当てられる)．そして結果が入手できしだいスロット内に移動します．

リオーダ・バッファからのフォワーディングには，バッファ内容の検索が必要です．条件付きのARM命令は，有効な結果をバッファに戻すこともあれば戻さないこともあります．複数の命令が，同じデスティネーション・レジスタを使うこともあります．特定のレジスタについて最も新しい値を見つけるには，結果があるスロットで有効になるのを待ち，無効になるのをみてそれから(最悪時)他の全スロットを順番に検索しなければならない場合があります．このような最悪の場合はまれですが，非同期タイミング・フレームワークでは簡単に実現できます．



〔図14.11〕AMULET3リオーダ・バッファ構成

◆レジスタ書き込み

リオーダ・バッファが生成するライトバック・ストリームはプログラム順になっているため，パイプラインのこのステージは，各値の有効性を確認し，メモリ・フォールトを探しながらレジスタ・ファイルに値を供給するだけです．フォールトを検出すると例外を発生し，例外処理メカニズムがアクティブになるまで後続の結果を廃棄します．

◆AMULET3の性能

AMULET3の性能的特徴は，表14.4のとおりです．同じプロセス上でARM9TDMIに匹敵する性能を得るという目的が達成されていることがわかります．

AMULET3は，DRACO通信コントローラの処理コアとして使用されていますが，これは次節で説明します．

〔表14.4〕AMULET3の特性

プロセス	0.35 μm	トランジスタ数	113,000	MIPS	120
金属レイヤ	3	コア面積	3 mm^2	消費電力	154 mW
V_{dd}	3.3 V	クロック	none	MIPS/W	780

14.6　DRACO通信コントローラ

DRACO (DECT Radio Communications Controller) チップは，AMULET技術をベースとした最初の商用設計です．このチップでは，計算および制御エンジンとして後で述べるAMULET3Hセルフタイム処理サブシステムを使用しています．

DRACOは，欧州連合 (EU) の資金援助を得てHagenuk GmbH (同期通信ペリフェラルを設計) とマンチェスター大学 (AMULET3Hサブシステムを設計) によって共同開発されました．

◆セルフタイム設計の根拠

DRACOチップにセルフタイム処理サブシステムを採用したおもな理由は，セルフタイム論理のほうが電磁環境適合性 (EMC) において有利だからです．つまり，高速クロック・システムが生成するラジオ障害は，DECTラジオ通信の性能を低下させる可能性があります．クロックの高調波がDECTチャネル周波数帯の一つに入ってしまう可能性が高く，そうなるとチャネルが使用できなくなってしまうのです．

論理的な結論としては，チップ全部をセルフタイム論理で動作させることになります．しかし，多くの通信インターフェースは本質的に同期的であり，同期合成ツールを使って設計するほうがずっと簡単です．

同期ペリフェラル・サブシステムは，セルフタイム処理サブシステムのもつEMCの有利さを妨げません．なぜならペリフェラルの動作周波数はすべて処理速度よりずっと低いからです．つまり，高速処理や大きな負荷のかかる外部メモリ・バスは全部，非同期で動作しているため，完全なセルフタイム・システムのもつ利点をほとんど備えつつ，セルフタイム通信ペリフェラルを開発する設計コストはかかっていません．

◆DRACOの機能

DRACOチップには，同期ペリフェラル・サブシステムに以下の機能をもっています．

(1) 16Kbps/秒 HDLCコントローラとトランスレス・アナログISDNインターフェースを備えたISDNコントローラ
(2) DECTラジオ・インターフェース・ベースバンド・コントローラおよびDECTラジオ・サブシステムへのアナログ・インターフェース
(3) DECT暗号化エンジン
(4) 4チャネル全二重ADPCM/PCM変換信号プロセッサ
(5) DECTコントローラがバッファ・スペースに使用する8Kバイトの共有RAM
(6) アナログ通信ポートとしても使用可能な音声入出力用アナログ・フロントエンド付き通信コーデック
(7) どちらのUARTが使ってもよいIrDAインターフェース付きの二つの高速UART
(8) 割り込みコントローラ
(9) カウンタタイマとウォッチドッグ・タイマ
(10) プログラム可能切り替え機能付き2MbpsIOM2ハイウェイ・コントローラ
(11) I^2Cインターフェース
(12) アナログ・ディジタル変換器 (ADC) インターフェース
(13) ハンドシェイク機能付き8ビット・パラレル・ポートを含む65のフレキシブルI/Oポート
(14) 二つの汎用パルス幅変調コントローラ
(15) オンチップ・オシレータは38.864MHzを生成，オンチップ・フェーズロック・ループはISDNインターフェースに必要な12.288MHzマスタ・クロックと，DECTコントローラに必要な13.824MHzマスタ・クロッ

クを生成

(16) ISDN-DECTシンクロナイザ・フェーズロック・ループが，ISDNとDECTのクロック・ドメイン間のデータ転送におけるビット・ロスを防止

(17) クロック・システムにパワーダウン機能を搭載

さらにAMULET3H処理サブシステムには，以下が組み込まれています．

(18) ARMアーキテクチャversion4T (Thumb16ビット圧縮命令セットを含む) を実現するデバッグ・ハードウェア付き100 MIPS AMULET3 32ビット・プロセッサ

(19) 8Kバイトのデュアルポート高速RAM (プロセッサにローカル)

(20) 同期ペリフェラルに接続する同期オンチップ・バスへのブリッジ付きセルフタイム・オンチップ・バス

(21) 32チャネルDMAコントローラ

(22) 標準通信アプリケーション・ソフトウェアを格納した16KバイトのROM

(23) A-D変換が完了する間，プロセッサをゼロ電力ウェイト・モードに保ち，それによってソフトウェアを外部データ速度に同期化する非同期イベント駆動ロード・モジュール

(24) SRAM，DRAM，フラッシュ・メモリの直接接続に対応するプログラム可能外部メモリ・インターフェース

(25) ソフトウェアによって校正される，オフチップ・メモリ・アクセス・タイミングを制御するオンチップ基準ディレイ・ライン

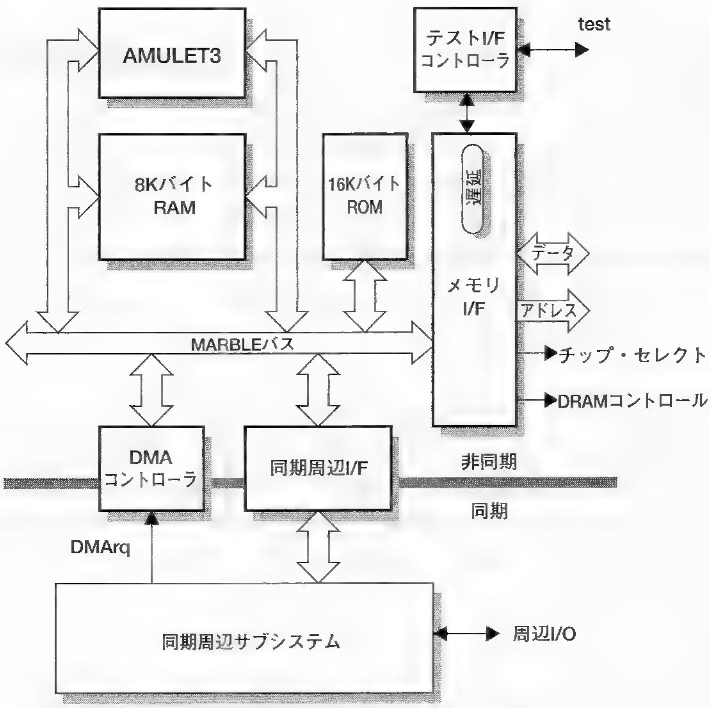
◆AMULET3H

AMULET3Hは，AMULET3コアをベースとしたサブシステムで，DRACO通信コントローラの中心にある非同期の「島」となって多様な同期ペリフェラル・コントローラにインターフェースしています．非同期動作の利点を生かすため，コアは非同期に動作するいくらかのメモリにアクセスする必要があります．そしてやはり非同期に動作するオフチップ・メモリをもつことは電磁適合性において大きな利点があります．

◆MARBLEオンチップ・バス

非同期サブシステムの構成は，図14.12のとおりです，AMULET3コアはデュアルポートRAM (下記に詳しく述べる) へ，そしてMARBLEオンチップ・バスへ直接接続されています．MARBLEは概念的にはARMのAMBAバスに似ていますが，主な違いはクロック信号を使わないことです．その転送メカニズムは分割トランザクション・プリミティブをベースとしています．

ローカルRAM以外のシステム・コンポーネントには，MARBLEバス経由でアクセスします．このようなコンポーネントには，オンチップROM，DMAコントローラ，特定アプリケーション向けペリフェラル・コントローラが存在する同期バスへのブリッジ，外部メモリへのインターフェースがあります．



〔図14.12〕AMULET3H非同期サブシステム

◆外部メモリ・インターフェース

外部メモリ・インターフェースは，オフチップ・デバイス用に従来の信号群を出します．これはAMULET2eインターフェース(14.4節参照)に似ていて，構成が非常に自由であり，外部アクセスと同期するのに基準ディレイを使用します．しかし，AMULET3Hにはオフチップ基準ディレイの代わりにオンチップ・ディレイがあり，これはタイミングの基準に対してソフトウェアで較正可能です．恐らくこの目的には，同期ペリフェラル・サブシステム内のタイマ/カウンタを使用します．

従来のROM，SRAM，メモリ・マップ済みペリフェラル・デバイスと同様，オフチップDRAMも直接サポートされています．

◆AMULET3Hメモリ構成

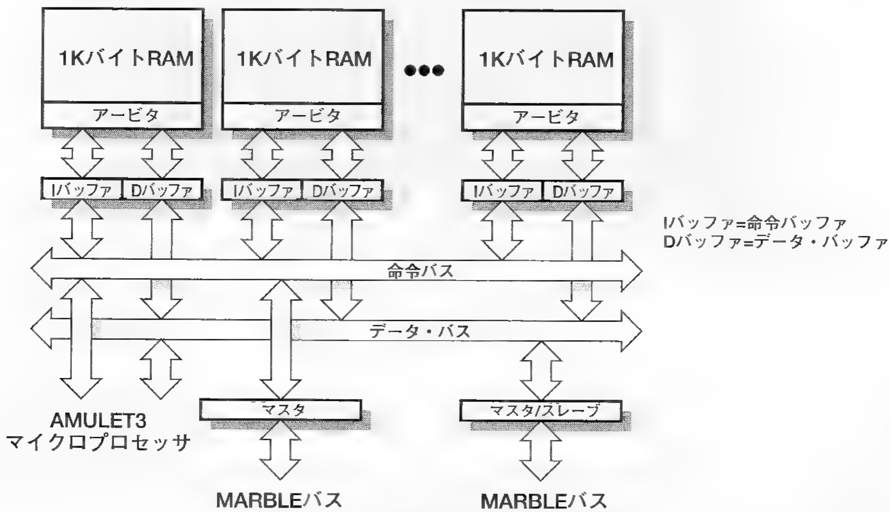
AMULET3プロセッサ・コアは，命令メモリ・アクセス用とデータ・メモリ・アクセス用に別々のアドレス・バスとデータ・バスをもっています．これには通常，別々の命令メモリとデータ・メモリが必要です．RISCシステムはよく，別々の命令キャッシュとデータ・キャッシュ，それに統合メイン・メモリを備えた「修正ハーバード」アーキテクチャを使用しています．

AMULET3Hコントローラは，キャッシュ・メモリではなくダイレクトマップRAMを使用しています．なぜならこのほうが経済的なうえ，リアルタイム・アプリケーションにおいては決まった動作をするからです．このコントローラは，デュアルポート・メモリ構造(14.13節参照)を使っているため，命令メモリとデータ・メモリ(それに，それらの整合性を保つ問題も回避)も分けていません(図14.13)．個別のビット・レベルでメモリをデュアルポートにするとコストがかかりすぎるため，メモリは1Kバイトのブロック八つに分かれてお

り，各ブロックに2ポートがあって内部的に調停が行われています．データ・アクセスと命令アクセスは，異なるRAMブロックに平行して起こっても互いに妨害されません．同じブロックで競合してしまった場合，片方のアクセスについては，もう片方が完了するまで待つために遅延がかかります．

各RAMブロックに別々の4ワード命令バッファとデータ・バッファを組込むことで，競合(と平均メモリ・アクセス時間)はさらに削減されています．ブロックへの各アクセスは，まず要求されたデータがバッファにあるかどうか確認します．そして，ない場合だけ，競合の危険を冒してRAMに取りに行かなければなりません．シミュレーションによると，命令フェッチの約60%が両バッファ内で満たされることになり，多くの短いタイムクリティカルなループは完全にバッファから動作することになります．

両バッファは事実上，RAMブロックの前の単純な128バイト第1レベル・キャッシュを形成しています．RAM配列を使用しないことによって読み出しサイクルが速くなること(これは非同期パイプラインによって自動的に受け入れられる)を考えた場合，これはまさに適切なたとえです．



〔図14.13〕AMULET3Hメモリ構成

◆テスト・インターフェース・コントローラ

AMULET3Hは商用に開発されつつあるため，生産時もテスト可能でなければなりません．これを可能にする機能は数多く設計に採用されていますが，もっとも顕著なのがテスト・インターフェース・コントローラ(図14.12参照)です．この外部メモリ・インターフェース論理に対する拡張は，外部メモリ・インターフェースの利用においてはAMBAによる例題セット(「テスト・インターフェース」参照)にならっていて，通常動作中はMARBLEのスレーブ，テスト目的ではMARBLEのマスタとなります．テスト・モードでは，製品テスト装置はMARBLEマスタとなり，オンチップROMの読み出しとオンチップRAMの読み出しおよび書き込みが可能です．また，AMULET3Hシステムが内蔵するその他多くのテスト機能(すべてMARBLEバスに接続したテスト・レジスタ経由で制御される)へもアクセス可能です．

◆AMULET3Hの性能

シミュレーションで得られたAMULET3Hの性能的特徴は，表14.5にまとめてあります(トランジスタ数，

面積と電力に関する数値は非同期サブシステムのみのもの．DRACOダイ全体は，7.8×7.3 mm)．オンチップRAMはプロセッサに必要なピーク命令帯域幅を与えることができないため，最大システム性能はAMULET3コア（表14.4参照）より若干低めです．またシステムの電力効率は，プロセッサだけの電力効率より低くなっています．これは，メモリ・システム電力もプロセッサ・コア自体が消費する電力とともに含まれるためです．

このように，AMULET3Hサブシステムは，性能と電力効率の点で相当する同期ARMシステムに匹敵しています．そのうえ，電磁干渉とシステム電力管理においては有利です．これらの長所は，現在の非同期設計向けのツール・サポート不足より重視されるべきです．

〔表14.5〕AMULET3Hの特性

プロセス	0.35 µm	トランジスタ数	825,000	MIPS	100
メタル層	3	コア面積	21 mm^2	消費電力	215 mW
V_{dd}	3.3 V	クロック	none	MIPS/W	465

◆DRACOアプリケーション

DRACOチップは，多数の通信アプリケーションで使用可能です．典型的なアプリケーションは，ISDNターミナルとDECTベースステーションを組み合わせたもので，これを通じて多くのユーザがコードレスDECTハンドセットを使って通信したり，ISDNラインまたはコードレスISDNネットワーク・ターミネータ経由で通信ネットワークに接続したりできます．チップの主なターゲット市場は，コードレスDECTデータ・アプリケーションにあります．

◆DRACOシリコン

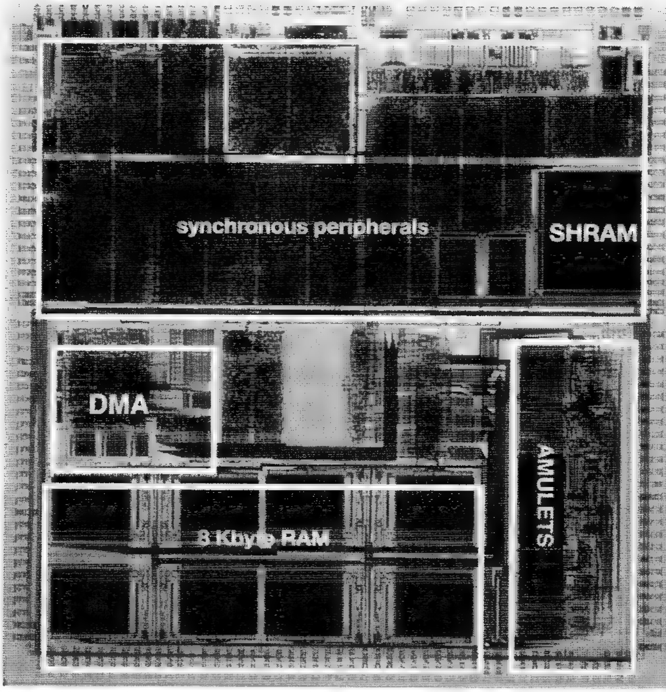
DRACOシリコンの主要な機能領域は，図14.14のとおりです．AMULET3H非同期サブシステムは，コア領域の下半分を占めています．また，同期通信ペリフェラルは上半分を占めます．最初のシリコンは，2000年前半に製造されました．

14.7 セルフタイムの将来

AMULET3はいずれ商業的に使用されるでしょうが，AMULETプログラムは，今のところはおもに研究用であり，AMULETコアが普及している同期型ARMコアに今すぐ取って代わる見込みはありません．しかし現在，非同期型設計スタイルは，電力を節約，電磁適合性を向上し，コンピュータ・ハードウェア設計に幅広いモジュラ式アプローチが可能になるとして世界中の注目を集めています．

グローバル・クロックを外し，実行する仕事があるときは各サブシステムが独自にタイミング機能を実行するようにすれば電力を節約することは理論的には明らかですが，商用的に意味があるほど複雑な回路で実際に利点が明らかになっている例はほとんどありません．AMULETの研究は，非同期型テクノロジの利点を具体的に証明することを直接的な目的としています．

セルフタイム設計スタイルが広く採用されるのを妨げているのは，現在の設計業界における知識基盤です．



〔図14.14〕DRACOダイ・プロット

多くのIC設計者は，初期の非同期型コンピュータの設計者たちが経験した困難のおかげで，非同期型に対して拒否するように教育されています．セルフタイム設計への無規律なアプローチからくる困難さと最近の進歩のおかげで，非同期型設計フレームワークが生まれました．これは，同期設計のフレームワークで提供されるものより無秩序的なアプローチではありますが，そこに受け継がれてきた問題をほとんど乗り越えたといえます．

あと数年のうちにわかることですが，世界中で行われているAMULETとそれに類似した開発が，非同期型の利点を証明すれば，設計者たちは過去の教育を忘れて新しい方法を学ばなければならなくなるでしょう．

◆AMULETへの支援

AMULET1は，Open Microprocessor systems Initiative _ Microprocessor Architecture Project (OMI-MAP) の一環として欧州共同体の資金援助で開発されました．AMULET1は，Open Microprocessor systems Initiative _ Deeply Embedded ARM (OMI-DE/ARM) プロジェクトの一環として開発されました．AMULET3とDRACOの開発は，OMI-DE2およびOMI-ATOMプロジェクトに出資した欧州連合内で支援されました．また研究の多くの部分は，ROPA奨学金GR/K61913の名目で博士号奨学金およびツール開発の形をとり，Engineering and Physical Sciences Research Council (EPSRC) を通じて英国政府から支援を受けました．

この研究は，ARM Limited, GEC Plessey Semiconductors (現在はMITELの一部)，VLSI Technology, Inc.からも，さまざまな形で支援を受けました．Compass Design Automation (現在はAvant!の一部) のツールは両プロジェクトの成功に大きく貢献し，EPIC Design Technology, Inc. (現在はSynopsysの一部) のTimeMillはAMULET2とAMULET3の正確なモデリングに非常に役立ちました．

14.8 例題と演習

<例題14.1> セルフタイム設計の長所と短所をまとめなさい．

ここで説明した長所は以下のとおりです．

- 電磁環境適合性（EMC）の向上
- 電力効率の向上
- モジュラ式設計の向上
- クロック・スキュー問題がない
- 最悪時性能でなく実際的な性能を得られる可能性

短所は以下のとおりです．

- セルフタイム設計用の設計ツールがない
- 設計者の経験不足
- 設計教育の中でセルフタイム・テクニックへの反感が奨励されてきたこと
- 上記長所が商業規模でアピールされることがない

一般に，設計上の問題に対してセルフタイム式のソリューションを追及することはまだハイリスクだと言えます．またツール不足によって非常に労力がかかる恐れもあります．

演習14.1.1 プロセッサ・コアに別々の命令ポートとデータ・ポートがあり，両方が一つのデュアルポート・メモリに接続している場合，メモリの競合がクロック式およびセルフタイム式プロセッサに与える性能的影響を推測しなさい．メモリは，AMULET3Hメモリと同様，調停される八つのセグメントで構成され，ライン・バッファがないものとします（「AMULET3Hのメモリ構成」参照）．

プロセッサは継続的に命令をフェッチし，約2命令につき1データ・メモリ・アクセスを必要とすると仮定します．クロック式プロセッサからの命令アクセスとデータ・アクセスは同期化されています（もちろんクロックによって）から，競合が起こるたびに2アクセスのうち一つが1サイクルの間ストールします．非同期プロセッサではそのような同期化がありませんから，二つのアクセスはランダムな相対タイミングをもち，ストール時間は，第2の競合アクセスが要求されたときに残っている最初の競合アクセス時間のうち何割かになります．

演習14.1.2 AMULET3Hメモリ・システムにある命令およびデータ・ライン・バッファが性能に与える影響を推測しなさい．前の演習と同じ想定をしてください．

付録 A コンピュータ論理について

●コンピュータ論理

コンピュータ設計は，線上の信号が**真**，**偽**いずれかの値をもつブール論理に基づいています．通常，グラウンドに近い電圧は「偽」，供給電圧に近いほうは「真」を指します．しかし，二つの状態を正確に反映できるなら，どのような表現も使用可能です．「真」は論理 ‘1’，「偽」は論理 ‘0’ と呼ばれることもあります．

●論理ゲート

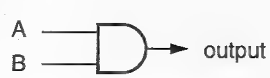
論理ゲートは，一つ以上の論理入力の機能を生成します．たとえば，2入力「AND」ゲートは，最初の入力が「真」かつ（AND）2番目入力が「真」なら「真」出力を生成します．各入力は「真」「偽」のいずれかですから，可能な入力の組み合わせは4種類しかありません．ゲートの全機能は図A.1のように**真理値表**として表すことができます．ANDゲートの論理記号も示してあります．

ANDゲートは，2入力以上にも拡張できます〔ただし，現在のCMOS技術では，実用上のファンイン（論理入力）は4入力に限っている．また，サブミクロン・プロセス技術のなかには3入力のものもある〕．全入力が ‘1’ のとき出力は ‘1’，少なくとも一つの入力が ‘0’ のとき出力は ‘0’ です．

ORゲートも同様に定義できます．全入力が ‘0’ のとき出力は ‘0’，少なくとも一つの入力が ‘1’ のとき出力は ‘1’ です．2入力ORゲートの真理値表と論理記号は図A.2に示します．

インバータも重要な論理要素です．これは，1入力で逆の出力を生成します．論理機能はNOTです．入力が「真」（‘1’）のとき，出力は「偽」（‘0’），またはその逆です．出力にインバータの付いたANDゲートはNAND（NOT AND）ゲートで，出力にインバータの付いたORゲートはNOR（NOT OR）ゲートです．反転はよくゲートの入力または出力に丸を付けて示します．

基本的なCMOSゲートは，元々インバータなので，じつはNANDはANDより単純に実現できます．ANDはNANDにインバータを付けたものなのです．同様にORゲートはNORゲートにインバータを付けたものです．



論理シンボル

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

真理値表

〔図A.1〕ANDゲートの真理値表と論理記号



A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

論理シンボル　　　真理値表

〔図A.2〕ORゲートの真理値表と論理記号

●ブール代数

どんな論理回路でも，たった二つのNANDゲートを使って構築できます．これは，ブール代数の規則に基づいています．まず，NANDゲートの両入力が同じ信号に接続している場合，出力は入力の反転ですから，これでインバータができます．次に，インバータをNANDゲートの各入力に接続します．真理値表を見れば，この回路がORの機能を果たしていることがわかります．

論理方程式を書くには，数学記号をよく使います．ANDは「・」，ORは「+」です．したがって「*A* AND (*B* OR *C*)」は「*A*・(*B*+*C*)」と書けます．論理反転は上線で表します．たとえば，$\overline{A \cdot B}$と書けます．文脈により，1+1=1のブール論理方程式であるということが明らかな場合，この表記法は非常に便利です．

●2進法の加算

コンピュータでは，数字は通常2進法表記で表します（データ型と数字表記については，6.2節を参照）．つまり，各桁が0から9の範囲をとり，位置が10の累乗の重みをもつ（1，10，100，・・・）使い慣れた10進法でなく，各桁が0または1で位置が2の累乗の重みをもつ（1，2，4，8，16，・・・）2進法を使います．2進法の各桁（ビット）は，ブール値で表すことができる二つの値のいずれかに限られるため，2進法の数字はすべてブール値で表すことができます．

二つの単一ビット・バイナリ数の加算は，前に述べた論理ゲートを使って行うことができます．0+0=0，1+0=1，1+1=2，つまり2進法では2ビットの10となります．したがって，二つの単一ビット入力の加算器は，出力として2ビットが必要です．つまり，入力ビットと同じ重みの**サム**（和）ビット，そしてその2倍の重みの**キャリ**（桁上げ）ビットです．

入力が*A*と*B*の場合，サムとキャリは以下のようになります．

$$sum = A \cdot \overline{B} + \overline{A} \cdot B \quad \text{（式18）}$$

$$carry = A \cdot B \quad \text{（式19）}$$

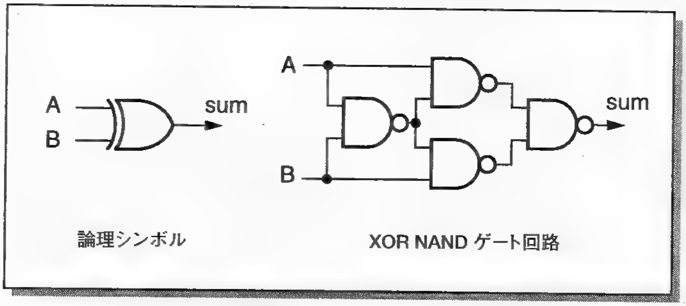
*sum*は，ディジタル論理によく登場し，**排他的OR**（exclusive OR）もしくはXOR関数と呼ばれます．「排他的」というのは，*A*が真，または*B*が真のとき真であり，両方とも真のときは真でないからです．排他的ORの論理記号を図A.3に示します．四つのNANDゲートを使った（わかりにくい）実装も添えてあります．

*N*ビットのバイナリ数の加算器は，単一ビットの加算器から作ることができます．ただし，1桁目以外は，すぐ下からのキャリインも受け入れなければなりません．加算器の各ビットは，サムおよび入力によるキャリイン，キャリアウトを生成します．

$$sum_i = A_i \cdot \overline{B}_i \cdot \overline{C}_{i-1} + \overline{A}_i \cdot B_i \cdot \overline{C}_{i-1} + \overline{A}_i \cdot \overline{B}_i \cdot C_{i-1} + A_i \cdot B_i \cdot C_{i-1} \quad (式20)$$

$$C_i = A_i \cdot B_i + A_i \cdot C_{i-1} + B_i \cdot C_{i-1} \quad (式21)$$

この方程式では，i=1～Nを適用し，C_0は0です．



〔図A.3〕論理記号とXORゲートのNAND回路

●マルチプレクサ

プロセッサ実装では，サイクルごとに多数の入力からオペランドのソースを選択しなければなりません．これを実行する論理コンポーネントがマルチプレクサ（もしくは**MUX**と記す）です．2入力マルチプレクサは，ブール式「選択」入力（S）と二つのバイナリ入力値Ai，Biをもちます．1≦i≦NのNは各バイナリ値のビット数を表します．Sが0のとき出力Z_iはA_iに等しく，Sが1のときZ_iはB_iに等しくなります．これは，直截的な論理関数です．

$$Z_i = \overline{S} \cdot A_i + S \cdot B_i \quad (式22)$$

●クロック

ほとんどのプロセッサは，**クロック**という自走タイミング基準信号により制御されています（すべてではない．外部タイミング基準信号なしで動作するAMULETプロセッサ・コアについては9.4節を参照）．クロックは，プロセッサ内の状態変化を制御します．

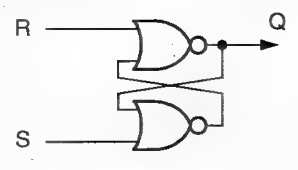
一般に，状態はすべて**レジスタ**に保持されます．クロック・サイクルの間に**組み合わせ論理**（進行中の入力値によってのみ出力が決まる論理）が，上記のようなブール論理ゲートを用いて次の状態値を割り出します．クロック・サイクルの終わりには，アクティブなクロック・エッジが，全レジスタを同時に次の状態にスイッチさせます．最高の性能を出すためには，クロック周波数はすべての組み合わせ論理が必ず完結する最高レートに設定し，最悪でも次のアクティブ・クロック・エッジまでには完結するようにします．

●シーケンス回路

レジスタは，アクティブなクロック・エッジ間で状態を保存し，その内容はアクティブ・エッジで変わります．これを**シーケンス回路**といい，出力は現在の入力値だけでなく，入力値が過去にどう変化したかにも依存します．

もっとも単純なシーケンス回路は**RS（リセット・セット）フリップフロップ**です．この回路では，「セット」入力がアクティブのとき，つねに出力を‘H’にセットし，「リセット」入力がアクティブのときはつねに出力を‘L’にリセットします．両入力が同時にアクティブな場合，フリップフロップの動作は実装に依存します．両方が非アクティブの場合，フリップフロップは最後の状態を「記憶」しています．二つのNORゲートを使っ

たRSフリップフロップについて図A.4にシーケンス表とともに示してあります．出力は現在の入力値の組み合わせ関数ではありませんから，これはもはやたんなる真理値表とはいえません．



回路

R	S	Q_{t+1}
0	0	Q_t
0	1	1
1	0	0
1	1	不定

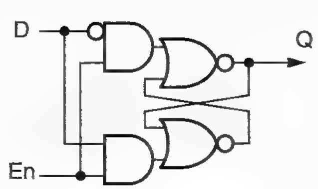
シーケンス表

〔図A.4〕RSフリップフロップ回路とシーケンス表

●トランスペアレント・ラッチ

トランスペアレント・ラッチ（または**D型ラッチ**）には，データ入力（*D*）とイネーブル信号（*En*）があります．*En*がハイのとき出力は*D*入力に従い，*En*が‘L’の間は*En*がローになる直前に適用した値で固定します．図A.5に示すように，RSフリップフロップを使って*D*と*En*から*R*と*S*を生成すれば作ることができます．

原則的には，ステージ間の組み合わせ論理が十分にゆっくりであれば，D型ラッチを使ってどんなシーケンス回路でも作成できます．*En*に非常に短い正パルスを送ると，次の状態データがラッチを通り，組み合わせ論理が新しい値に反応する前にホールドされます．しかし実用上，これで信頼性の高い回路を作るのは困難です．



回路

En	D	Q_{t+1}
0	0	Q_t
0	1	Q_t
1	0	0
1	1	1

シーケンス表

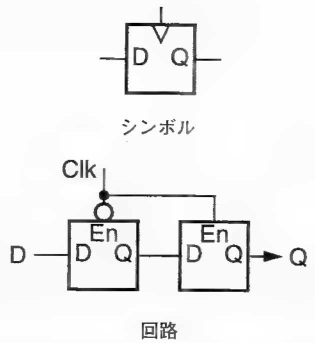
〔図A.5〕 D型トランスペアレント・ラッチ回路とシーケンス表

●エッジ・トリガ・ラッチ

もっと信頼性の高いラッチ回路を作る方法は多くありますが，クロック・サイクルごとに各信号が二つのトランスペアレント・ラッチを通過することが必要です．もっとも単純な形では，2番目のラッチが最初のラッチの後に並び，反転したイネーブルで動作します．つねに，片方のラッチが保持し，片方はトランスペアレントです．クロック・エッジのたびに，最初のラッチがトランスペアレントでなく，2番目がトランスペアレントになり，入力データ値が出力へと伝わります．そして，次のサイクルの同じエッジがくるまで保持します．これが**エッジ・トリガ・ラッチ**です．論理記号，回路，シーケンス表は図A.6のとおりです（表で，入力欄のxは，入力が出力に影響しないことを表す）．

エッジ・トリガ・ラッチの設計には神経を使います．なぜなら，たんなる組み合わせ論理回路ではなく，そ

の機能はほとんど回路要素の動的性質によって決まるからです．二つのトランスペアレント・ラッチを並べて上記のように作った場合，避けるべき「競合状態」が多くあります．しかし，良いツールを使えば信頼性の高いラッチを作ることは可能であり，信頼性の高いラッチがあれば任意の複雑さのシーケンス回路が作れます．



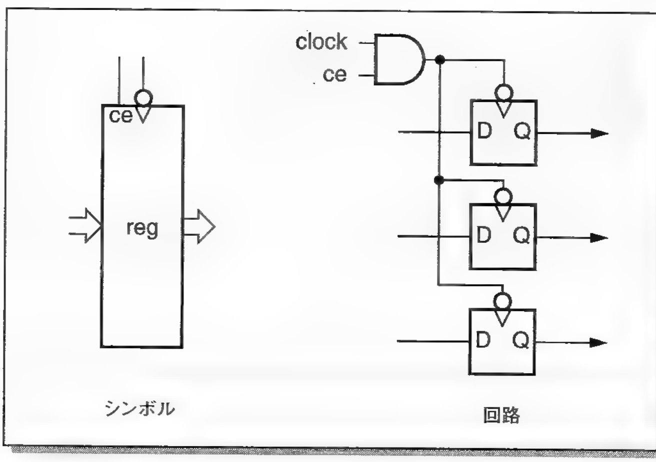
Clk	D	Q_{t+1}
0	x	Q_t
1	x	Q_t
↑	0	0
↑	1	1
↓	x	Q_t

シーケンス表

〔図A.6〕 D型のエッジ・トリガ・ラッチ記号，回路，シーケンス表

●レジスタ

クロック・サイクル中にバイナリ値を保存する1セットのエッジ・トリガ（もしくは同等の）ラッチをレジスタと呼びます．柔軟なレジスタはフリーラン・クロックに接続し，「クロック・イネーブル」制御入力をもつため，制御論理はサイクルごとにレジスタの内容を更新するかどうか決定できます．そのようなレジスタを簡単に作るには，エッジ・トリガ・ラッチの共通クロック線にゲートを加えます．負エッジ・トリガ・ラッチを使えば，ANDゲートは高クロック・パルス全体を削除します．ただし，イネーブル入力は，クロックが‘H’になる前に‘L’であり，クロックが再度‘L’になるまで‘L’でなくてはなりません（これらはイネーブル入力の**セットアップ条件**と**ホールド条件**と呼ばれる）．イネーブル自体が同じクロックの同様のレジスタから生成したものであれば，これらの制約が満たせます．レジスタ回路は図A.7のとおりです．



〔図A.7〕 クロック・イネーブル制御信号をもつレジスタ

付録

B 用語解説

● **ACM**

Association for Computing Machinery．コンピュータ専門家でつくる米国の学会．

● **Acorn Computers Limited**

1983-1985年にARMプロセッサを開発した英国の会社．英国でコンピュータの教育現場での利用を普及させたBBCマイクロコンピュータも開発．BBCマイクロに続きARMベースの一連のArchimedesコンピュータも学校市場に投入した．

● **ADC**

Analogue to Digital Converter（A-Dコンバータ）．一定の範囲内で変化する入力を取り込んで，入力を2^nの離散値のうち一つに近似されたnビットの2進数に変換する電子回路．

● **ALU**

Arithmetic-Logic Unit．算術論理演算装置．プロセッサの構成要素で，演算（加減，場合により乗除），論理（シフト，ビットでのANDなど）演算を行う．

● **AMBA**

Advanced Microcontroller Bus Architecture．複雑な組み込み型システムチップを作るうえで多様なモジュールを接続するオンチップ・バスのオープン規格．

● **AMULET**

Asynchronous Microprocessor Using Low-Energy Technology．マンチェスター大学で開発されたARMアーキテクチャの非同期実装プロトタイプを指す．

● **ANSI**

American National Standards Institute．米国規格協会．ASCII，ANSI標準Cなどコンピュータで広く使用する多くの規約を定めた．

● **APCS**

ARM Procedure Call Standard. ARM Limitedが定めた呼び出し規則で，異なるコンパイラで生成した（またはアセンブリ言語で書かれた）プロシージャが互いにコールできる．

● ARM（旧ARM社）

旧Advanced RISC Machine（その前はAcorn RISC Machine）社．しかし，単語を綴った名前は現在，正式に廃止された．RISC設計原理に基づく（RISC参照）32ビット・マイクロプロセッサ．

● ARM Limited（ARM社）

旧社名Advanced RISC Machines Limitedの略称であったが，現在では上記が正式社名．1990年Acornから独立，ARMテクノロジを開発した会社．英国ケンブリッジ本社だが，設計および販売活動は世界複数の地域で行われている．

● ASCII

American Standard for Computer Information Interchange. 米国規格協会が制定した情報交換用標準符号．プリント可能文字およびプリント制御文字を7ビットのバイナリ数で表す標準的な方法（最近のコンピュータでは，8ビットのフィールドにまで拡張することも多い）．

● ASIC

Application-Specific Integrated Circuit. 特定のアプリケーション向けに設計されたVLSIデバイス．通常は特定の顧客向け．

● ASSP

Application-Specific Standard Part. 特定のアプリケーション向けに設計されたVLSIデバイスで，半導体メーカが標準コンポーネントとして販売する．ASICとして開発されたチップで市場の要求が高いものを，メーカと元々の顧客が合意を結ぶことによって一般の顧客も入手できるようにしたもの．

● BBC

The British Broadcasting Corporation. 英国の公共ラジオ・テレビ放送会社．ライセンス料金を資金源としているが，英国ではこれがテレビ所有権の義務である．BBC憲章には一般市民の教育が含まれ，1982年に「The Computer Programme」と題した人気シリーズを製作，同シリーズのために委託したBBCマイクロを取り上げた．

● BCD

Binary Coded Decimal. 2進化10進数．10進数の各桁をコード化し，同等の2進法として表す方法．

● BCS

The British Computer Society. 英国のコンピュータ専門家でつくる学会．

● C

The C programming language. 汎用および組み込みシステム開発に広く使用されているプログラム言語．

● CAM

Content Addressable Memory. 多くのデータ項目を含み，データ値を提示するとそれらすべてと比較して

一致するものがあるかどうか探す．一致するものがあれば，その位置アドレスが出力され，なければCAMは「ミス」を出力する．「連想メモリ」とも呼ばれ，連想キャッシュもしくはTLBの重要なコンポーネントともなっている．

●CISC

Complex Instruction Set Computer．複雑な命令セット・コンピュータという用語は，RISCの性質をもたないアーキテクチャを表すため，RISCと同時に作られた．典型的なCISCは，可変長命令ミニコンピュータ型の命令セットをもち，多数のアドレシング・モード，メモリからメモリへの操作があり，多くのデータ型に対応する．

●CMOS

Complementary Metal Oxide Semiconductor．相補型金属酸化膜半導体．現在のICに使われる一般的な技術．CMOSは同じチップ上でNMOSとPMOS FFTを組み合わせることによって，両方向にアクティブな信号駆動ができ，スイッチングが高速で，スイッチングしていない時は電力損失がほぼゼロになる．

●Codec（コーデック）

Coder-decoder（コーダ＆デコーダ）．入力（たとえばアナログ音声信号）をディジタル形式に変換する電子システム．また，この変換の逆も可能．Compressor-decompressorの略語としても使用される．

●CPI

Cycles Per Instruction．プロセッサの効率の単位で，クロック・サイクル数を典型的なコード・シーケンス中の命令数で割る．

●CPSR

Current Program Status Register．ステータス・コード・ビット，割り込みディセーブル・フラグ，プロセッサ動作モード・ビットを含むARMレジスタ．

●CPU

Central Processing Unit．中央演算処理装置．コンピュータ内のプロセッサを表す，多少あいまいな用語．プロセッサ・コアだけを指す場合もあれば，オンチップMMUとキャッシュ，メイン・メモリを含むこともある．本書ではキャッシュ・メモリを含むプロセッサ・コアのみを指し，プロセッサ，キャッシュ，MMU（あれば）で構成されるシステムを表現するのに使用する．

●DAC

Digital to Analogue Converter（D-Aコンバータ）．通常n本のバイナリ形式で表現されるディジタル信号を，1本の線の2^n値のうち一つの値をもつアナログ信号に変換する電子回路．

●DECT

Digital European Cordless Telephone．発話がディジタル形式で受話器からラジオ・リンクを通じて地方局へ送られるコードレス電話に関する欧州の規格．

●DRAM
Dynamic Random Access Memory．ビット単位がもっとも安いRAM．多くのコンピュータ・システムで主メモリとして使用している．データはコンデンサに電荷として保存され，数ミリ秒で消える（したがって「動的」）．長時間保存するには，DRAMは定期的に「リフレッシュ」する必要がある．つまり，フル電荷を保存するには，まだ残っている間に読み，再度書き込まなければならない．

●DSP
Digital Signal Processor．ディジタル・データの連続ストリームを処理するために最適化された構成をもつプログラム可能プロセッサ．制御（意思決定）動作向けに最適化されたARMのような汎用プロセッサと対比される．

●EDO
Extended Data Out．DRAMメモリ用のインターフェース規格で，高性能メモリ・システム構築を可能にする．

●EEPROM（またはE^2PROM）
Electrically Erasable and Programmable Read Only Memory．適切な電気信号を与えることによってプログラムしたり消去したりできるROM．フラッシュ・メモリと似ているが異なる技術を使っており，一般にフラッシュより柔軟性が低い．

●EPLD
Electrically-Programmable Logic Device．オンチップ・メモリ・セルの状態によって接続が決まる多数のゲートをもった汎用論理チップ．これらのメモリ・セルを再プログラムしてデバイスの論理機能を変えることも可能（1回だけ設定可能なデバイスであるFPGAの項も参照）．

●EPROM
Electrically-Programmable Read Only Memory．適切な電気信号によってプログラム可能なROM．通常，何度も消去，再プログラムを繰り返す．

●Flash
EEPROMの一種で，電子的に消去および再プログラム可能．あまり頻繁に変更されないプログラムやデータを携帯システムに保存するのに広く使用されており，不揮発性ファイル・メモリの形で普及しつつある．

●FPA
Floating-Point Accelerator．浮動小数点処理を高速化する追加ハードウェア．ARM FPA10など．

●FPASC
Floating-Point Accelerator Support Code. ARMシステム上で動作するソフトウェアで，FPA10浮動小数点アクセレレータを含む．FPA10はハードウェアでARM浮動小数点命令セットのサブセットを実装し，残りの命令を扱うためにFPASCを必要とする．

● FPE

Floating-Point Emulator. FPA10浮動小数点アクセラレータを含まない，ARMシステム上で動作するソフトウェアで，ARM浮動小数点命令セットを処理する．

● FPGA

Field-Programmable Gate Array. アンチヒューズなど1回だけプログラム可能なオンチップ・コンポーネントの状態によって接続性が決まる多数のゲートをもった汎用論理チップ．現在では再プログラム可能なデバイス（EPLDとも呼ぶ）も含まれる．

● FPSR

Floating-Point Status Register. ARM浮動小数点アーキテクチャ内のユーザ可視レジスタで，多様な動作を制御し，エラー状態を表示する．

● FPU

Floating-Point Unit. プロセッサ内で浮動小数点演算処理を行うコンポーネント．

● FSM

Finite State Machine. 有限ステートマシン．内部状態をもち，出力と次の状態が入力と現在の状態の組み合わせ論理関数になっている順序ディジタル回路．

● GSM

Global System for Mobile communications. 欧州，アジア，その他の地域で広く使用されている携帯テレフォニのディジタル規格．

● IC

Integrated Circuit. 数個から100万個のトランジスタが同じ製造工程でプリントされた半導体デバイス．「チップ」と呼ばれることもある．

● IDE

Integrated Drive Electronics. すべての高度な機能がドライブ上にあるハード・ディスク・ドライブ用のインターフェース．ホスト・コンピュータ・インターフェースはデータ・バスと数本のアドレス線で構成する．

● IEE

The Institution of Electrical Engineers. 英国の電子（および電気）工学技術者でつくる学会．コンピュータ分野で，セミナー主催，雑誌の発行も行っている．

● IEEE

The Institute of Electrical and Electronics Engineers, Inc.. 米国の電子・電気工学技術者でつくる学会．会員は米国民に限らず，コンピュータ分野における国際会議主催や雑誌発行に力を入れている．ACMと協力する場合も多い．

● I²C

Inter-IC. 複数のICをプリント回路基板上で接続するためのシリアル・バス規格. システムの他の部分が電源オフになっていても起動されている小さなCMOS RAMやRTCチップを接続するのにとくに適している.

● I/O

Input / Output. コンピュータと環境の間の周辺デバイスを通じたデータ伝送.

● IrDA

Infra-red Data Association. 赤外線データ協会. 赤外線通信プロトコルを業界内で標準化するために設立された組織. 頭文字は, その規格に準拠したシステム・インターフェースによく使用される.

● ISDN

Integrated Services Digital Network. 発話を64Kbps秒のデータ・ストリームで送るディジタル・テレフォニに関する国際規格. 制御プロトコルも定めている.

● JTAG

Joint Test Action Group. ARMチップの多くで使用しているシリアル・インターフェースに基づくテスト標準を定めた委員会.

● LCD

Liquid Crystal Display. 軽量性と低消費電力によって, ラップトップ・コンピュータやPDAなど大半の携帯アプリケーションに使用されるディスプレイ技術.

● LRU

Least-Recently Used. アソシアティブ・キャッシュやTLBの中で新しい値の場所を空けるためにどの値を追い出せばよいかを選択するアルゴリズム.

● MFLOPS

Millions of Floating-Point Operations Per Second. 浮動小数点処理をする際のコンピュータ性能の単位.

● MIPS

Millions of Instructions Per Second. プロセッサが自分の命令を実行できる速度の単位. 命令セットにより各命令の意味的内容が異なるため, 二つのプロセッサをMIPSで比較することは意味がない. 各種ベンチマーク・プログラムで有効な比較が試みられており, またDhrystone MIPSも正規化された測定方法として有効と(議論はあるが)いわれている.

● MMU

Memory Management Unit. 仮想アドレスを物理アドレスに変換するプロセッサの一部で, メモリ内のページ・テーブルを使用する. テーブル・ウォーキング・ハードウェアやTLBを含む.

● Modem

Modulator-demodulator．モデム．ディジタル・データを（たとえば）従来の音声電話接続経由で送信可能な形式に変換する電子システムまたはサブシステム．同様に，受信した音声信号をディジタル・データに戻すことも可能．単純なモデムでは0について一つの音声周波数を，1について別の周波数を使用するが，現在の高速モデムではもっと高度な変調テクニックが使用されている．

● NMOS

N-type Metal Oxide Semiconductor. CMOS以前の半導体技術で，8，16ビットのマイクロプロセッサを作るのに使用された．アクティブなプルダウンおよびパッシブなプルアップ出力で，ゲートはスイッチングしていない時も低出力ドロー電流をもつ．NMOSトランジスタはPMOSトランジスタと組み合わせてCMOSを形成する．NMOSはPMOSより効果的なため，マイクロプロセッサにとってはNMOSがより好ましい技術とされているが，CMOSとして両方を組み合わせると，製造は複雑になるものの，速度と電力効率の点で大幅に優れている．

● OS

Operating System．オペレーティング・システム．アプリケーション・コードを管理し，スケジューリング，リソース割り当て，保護などを処理するシステム内の中心的コード．

● PC

Program Counter．次にフェッチする命令のアドレスを保持するプロセッサ内のレジスタ．次に述べるPCとは，文脈で区別する．

● PC

Personal Computer．明らかに商品名だが，いまやIBM PCと互換性のあるデスクトップ・コンピュータの総称として使われ，とりわけIntel x86命令セット・アーキテクチャをもつプロセッサを組み込んだものを指す．

● PCMCIA

Personal Computer Memory Card International Association. PC（Apple Newtonなど，他の携帯機器を含む）に差し込むカードの物理形状とインターフェース規格を管理する組織．名称に反して，メモリ・カードにかぎらず，周辺インターフェース・カードも標準化する．PCMCIAカードは，PCカードと略されることが多い．

● PDA

Personal Digital Assistant．個人向け携帯情報端末．最初はApple Newtonを指すよう作られた言葉だが，現在はパームトップ・コンピュータすべてについて使用する．

● PLA

Programmable Logic Array．複雑な多出力組み合わせ論理関数を実装する集積回路上の擬似正則構造．

● PLL

Phase-Locked Loop. 別のクロック信号を基準として使ってクロック信号を生成する回路.

● PMOS

P-type Metal Oxide Semiconductor. CMOSとNMOS以前の半導体技術で，初期の4ビットのマイクロプロセッサを作るのに使用された. アクティブなプルアップおよびパッシブなプルダウン出力で，ゲートはスイッチングしていない時も高出力ドロー電流をもつ. PMOSトランジスタはNMOSトランジスタと組み合わせてCMOSを形成する.

● PSR

Program Status Register. ステータス・コード，割り込みディセーブル・ビット，動作モード・ビットなどの情報を含むビットを保持するプロセッサ内のレジスタ. ARMはCPSR (Current Program Status Register) および各非ユーザ・モード用SPSR (Saved Program Status Register) を備えている.

● PSU

Power Supply Unit. 通常は商用電源やバッテリ電源を使ってシステムに必要な一定の供給電圧を提供する電子機器.

● RAM

Random Access Memory. じつはROMもランダム・アクセスなので，誤称といえる. RAMは，コンピュータにプログラムやデータを保存するのに使用する読み書きメモリを指す. このメモリを作るのに用いる半導体部品を意味することもあり，キャッシュやTLBなどの構造でも使われる.

● RISC

Reduced Instruction Set Computer. 縮小命令セットコンピュータ. 1980年ごろ，Patterson (U. C.Barkeley)，Ditzel (Bell Laboratories)，Hennessy (Stanford University) の3人が考え出した発想に基づいた，特定の性質をもつアーキテクチャのプロセッサ.

● ROM

Read Only Memory. コンピュータ内の決まったプログラム. この役割を果たす半導体デバイスも指す. RAMと対比される.

● RS232

モデム，プリンタ，他機器と接続して通信を可能にする非同期シリアル通信向けの規格.

● RTC

Real-Time Clock. コンピュータが時間，日付などを割り出すためのクロック・ソース. 通常，低周波数の水晶発振器を内蔵したバッテリ・バックアップされたシステムで，コンピュータ自体の電源が切れていても常に動作する.

● RTL

Register Transfer Level．プロセッサのようなハードウェア・システムの抽象概念．複数ビット値がバスに沿ってレジスタ間を流れると見なされる．

● RTOS

Real-Time Operating System．外部タイミングの制約を満たさなければならないプログラムをサポートするオペレーティング・システム（OS）．小型で（数Kバイト）組み込みシステムに適していることが多い．

● SDLC

Synchronous Data Link Controller．コンピュータをクロック式シリアル・インターフェースに接続し，一つ以上の標準プロトコルをサポートするペリフェラル．

● SoC

System-on-Chip．プロセッサ，メモリ，ペリフェラルなど電子システムの全機能を組み込んだ一つの集積回路．本書執筆現在では，大半のSoCがまだオンチップ・メモリに加えてオフチップ・メモリ・リソースを必要としているが，その他のシステム・コンポーネントは全部チップ上にある．

● SPSR

Saved Program Status Register．例外発生時にCPSRの値を保存するのに用いるARMレジスタ．

● SRAM

Static Random Access Memory. DRAMより高価だが，データをフリップフロップに保存するためリフレッシュが不要．DRAMよりアクセス時間が小さく，ほとんど電力損失なく永久に内容を保持することができる．キャッシュやTLBメモリなどプロセッサ・チップ上の多くのRAM機能に用いられ，小型の組み込みシステムの主メモリとしても使用される．

● TLB

Translation Look-aside Buffer．最近使ったページ・テーブル・エントリのキャッシュで，メモリ・アクセスのたびごとにページ・テーブル・ウォーキングのオーバヘッドがかかるのを防ぐ．

● UART

Universal Asynchronous Receiver / Transmitter．非同期シリアル通信用送受信回路．シリアル線（通常RS232信号プロトコル）とプロセッサ・バスのインターフェースとなる周辺デバイス．

● USB

Universal Serial Bus．さまざまなペリフェラルの接続に対応した最近のPCに搭載されている標準インターフェース．高速シリアル・プロトコルに加え，マシンの動作中にデバイスを接続したり外したりできる（つまり「ホットプラグ」可能）電気インターフェースを使用している．

● VHDL

VHSIC Hardware Description Language (VHSICはVery High-Speed Integrated Circuitの略). ハードウェアを動作レベルまたは構造レベルで記述するための標準言語で, 多くの半導体設計ツール会社にサポートされている.

● VLSI

Very Large Scale Integration. 超大規模集積. 多数のトランジスタを単一のチップに載せる製法. SSI, MSI, LSI (順にSmall, Medium, Large Scale Integration), VLSIなど, 規模ごとにチップ・トランジスタを分類する試みがあったが, 加工技術は新しい用語が作られるより速く進歩を遂げた. 次の用語ULSI (Ultra Large Scale Integration) は広く使用されるには至らず, おそらく現在ではすでに時代遅れになっていると思われる.

● VLSI Technology Inc.

たんにVLSIと略されることもある. Acorn Computersが設計した最初のARMチップを製造. 1990年, Acorn, AppleとともにARM Limitedを設立した. VLSIはARMの最初の半導体パートナとして数々のARMベースのCPUおよびシステム・チップを製造している. 現在はPhilips Semiconductorsの一部門.

● VM

Virtual Memory (仮想メモリ). プログラムが動作するアドレス空間で, MMUにより物理メモリに変換される. 仮想メモリは物理メモリより大きいこともあり, 仮想空間の一部をハード・ディスクにページ・アウトすることもあるが, どこにも実在しないこともある.

● VRAM

Video Random Access Memory. オンチップ・シフト・レジスタをもつDRAMの形式の一つ. シーケンシャル・データに対して高帯域幅でアクセスし, ビデオ・ディスプレイを生成する.

付録

C 参考文献

● ARMテキスト

(1) http://www. arm. com/

ARMプロセッサについてのデータシート，その他の関連資料はARM社のWebサイトからインターネット経由で入手できます．

(2) Jaggar（編）, ***ARM Architectural Reference Manual***, Prentice Hall. ISBN 0-13-736299-4.

ARMおよびThumb命令セット，メモリ管理アーキテクチャなどを詳しく説明した参考書．ARMコミュニティでは'ARM ARM' として知られています．

● Thumb命令について

(3) Segars, Clarke and Goudge, ***Embedded Control Problems, Thumb, and the ARM7TDMI***, IEEE Micro, 15(5)，1995年10月，pp. 22-30.

Thumb命令セットの思想と性能を記述した論文．

● RISCアーキテクチャについて

(4) Patterson and Ditzel, ***The Case for the Reduced Instruction Set Computer***, ACM SIGARCH Computer Architecture News, 8(6)，1980年10月，pp. 25-33.

プロセッサ設計の継続的な複雑化傾向に疑問を投げかけたゼミ論文．

(5) Katevenis, ***Reduced Instruction Set Computer Architectures for VLSI***, MIT Press，ケンブリッジ市，マサチューセッツ州，米国，1985年．ISBN 0-262-11103-9.

バークレーのRISC設計の詳細．ACM Doctoral Dissertation Award受賞．

(6) Hennessy and Patterson, ***Computer Architecture - A Quantitative Approach***, 2 nd. Ed. , Morgan Kaufmann，サンフランシスコ市，カリフォルニア州，米国，1990年．ISBN 1－55860-329-8.

RISC設計について権威ある書物．RISC誕生に最も貢献した2人による著作です．

(7) Patterson and Hennessy, ***Computer Organization and Design - The Hardware/Software Interface***, Morgan Kaufmann，サンフランシスコ市，カリフォルニア州，米国，1994年．ISBN 1-55860-281-X.

有名な「Computer Architecture - A Quantitative Approach」著者（順番が変わっている！）の大成功の

あと，詳細に入るというよりは基本に戻った書.

● CMOS設計について

(8) Weste and Eshraighan, *"Principles of CMOS VLSI Design, A System Perspective"*, 第2版, Addison－Wesley, Reading, マサチューセッツ州, 米国, 1993年. ISBN 0-201-53376-6.

トランジスタ・レベルのCMOS設計における権威のある一冊. 物理的レイアウト，回路，システムについて触れています.

● セルフタイム論理について

(9) 'Proceedings of the IEEE', 87 (2), 1999年2月, ISSN 0018-9219.

この非同期設計についての特別号には，バックグラウンド資料とAMULET2eなどいくつかの複雑な非同期設計に関する詳細が記されています.

(10) http://www.cs.man.ac.uk/async/

非同期論理のホームページで，チュートリアルや世界で非同期論理研究に従事するほとんどの団体へのリンクがあります.

● エンディアンについて

(11) Cohen, ***On Holy Wars and a Plea for Peace***, Computer, 14 (10), 1981年10月, IEEE Computer Society Press, pp. 48-54

バイト配列におけるビッグ・エンディアン，リトル・エンディアンの問題を取り上げた記事. ガリバー旅行記になぞらえてエンディアンという用語を定着させました.

● C言語について

(12) Kernighan and Ritchie, ***The C Programming Language***, 2 nd Ed. , Prentice-Hall, イングルウッドクリフ市，ニュージャージー州，米国, 1988年. ISBN 0-13-110362-8.

C言語に関する標準的参考書. 第2版では，第1版で触れていなかったANSI標準Cにも言及しています.

(13) ***Koenig, C Traps and Pitfalls, Addison-Wesley, Reading,*** マサチューセッツ州，米国, 1989年. ISBN 0-201-17928-8.

Cプログラミングによくある落とし穴を避ける方法を説明した書. どんなレベルのCプログラマにとっても有用です.

付録D 命令セット・リファレンス

● ARM命令セット1

Key to Tables	
{cond}	Refer to Table **Condition Field {cond}**
<Oprnd2>	Refer to Table **Operand 2**
<fields>	Refer to Table **PSR fields**
{S}	Updates condition flags if S present
C*, V*	Flag is unpredictable after these instructions in Architecture v4 and earlier
Q	Sticky flag. Always updates on overflow (no S option). Read and reset using MRS and MSR
x,y	B meaning half-register [15:0], or T meaning [31:16]
<immed_8r>	A 32-bit constant, formed by right-rotating an 8-bit value by an even number of bits
<immed_8*4>	A 10-bit constant, formed by left-shifting an 8-bit value by two bits

<a_mode2>	Refer to Table **Addressing Mode 2**
<a_mode2P>	Refer to Table **Addressing Mode 2 (Post-indexed only)**
<a_mode3>	Refer to Table **Addressing Mode 3**
<a_mode4L>	Refer to Table **Addressing Mode 4 (Block load or Stack pop)**
<a_mode4S>	Refer to Table **Addressing Mode 4 (Block store or Stack push)**
<a_mode5>	Refer to Table **Addressing Mode 5**
<reglist>	A comma-separated list of registers, enclosed in braces ({ and })
{!}	Updates base register after data transfer if ! present
§	Refer to Table **ARM architecture versions**

Operation		§	Assembler	S updates	Q	Action	Notes
Move	Move		MOV{cond}{S} Rd, <Oprnd2>	N Z C		Rd := Oprnd2	
	NOT		MVN{cond}{S} Rd, <Oprnd2>	N Z C		Rd := 0xFFFFFFFF EOR Oprnd2	
	SPSR to register	3	MRS{cond} Rd, SPSR			Rd := SPSR	
	CPSR to register	3	MRS{cond} Rd, CPSR			Rd := CPSR	
	register to SPSR	3	MSR{cond} SPSR_<fields>, Rm			SPSR := Rm (selected bytes only)	
	register to CPSR	3	MSR{cond} CPSR_<fields>, Rm			CPSR := Rm (selected bytes only)	
	immediate to SPSR	3	MSR{cond} SPSR_<fields>, #<immed_8r>			SPSR := immed_8r (selected bytes only)	
	immediate to CPSR	3	MSR{cond} CPSR_<fields>, #<immed_8r>			CPSR := immed_8r (selected bytes only)	
Arithmetic	Add		ADD{cond}{S} Rd, Rn, <Oprnd2>	N Z C V		Rd := Rn + Oprnd2	
	with carry		ADC{cond}{S} Rd, Rn, <Oprnd2>	N Z C V		Rd := Rn + Oprnd2 + Carry	
	saturating	5E	QADD{cond} Rd, Rm, Rn		Q	Rd := SAT(Rm + Rn)	No shift/rotate.
	double saturating	5E	QDADD{cond} Rd, Rm, Rn		Q	Rd := SAT(Rm + SAT(Rn * 2))	No shift/rotate.
	Subtract		SUB{cond}{S} Rd, Rn, <Oprnd2>	N Z C V		Rd := Rn - Oprnd2	
	with carry		SBC{cond}{S} Rd, Rn, <Oprnd2>	N Z C V		Rd := Rn - Oprnd2 - NOT(Carry)	
	reverse subtract		RSB{cond}{S} Rd, Rn, <Oprnd2>	N Z C V		Rd := Oprnd2 - Rn	
	reverse subtract with carry		RSC{cond}{S} Rd, Rn, <Oprnd2>	N Z C V		Rd := Oprnd2 - Rn - NOT(Carry)	
	saturating	5E	QSUB{cond} Rd, Rm, Rn		Q	Rd := SAT(Rm - Rn)	No shift/rotate.
	double saturating	5E	QDSUB{cond} Rd, Rm, Rn		Q	Rd := SAT(Rm - SAT(Rn * 2))	No shift/rotate.
	Multiply	2	MUL{cond}{S} Rd, Rm, Rs	N Z C*		Rd := (Rm * Rs)[31:0]	
	accumulate	2	MLA{cond}{S} Rd, Rm, Rs, Rn	N Z C*		Rd := ((Rm * Rs) + Rn)[31:0]	
	unsigned long	M	UMULL{cond}{S} RdLo, RdHi, Rm, Rs	N Z C* V*		RdHi,RdLo := unsigned(Rm * Rs)	
	unsigned accumulate long	M	UMLAL{cond}{S} RdLo, RdHi, Rm, Rs	N Z C* V*		RdHi,RdLo := unsigned(RdHi,RdLo + Rm * Rs)	
	signed long	M	SMULL{cond}{S} RdLo, RdHi, Rm, Rs	N Z C* V*		RdHi,RdLo := signed(Rm * Rs)	
	signed accumulate long	M	SMLAL{cond}{S} RdLo, RdHi, Rm, Rs	N Z C* V*		RdHi,RdLo := signed(RdHi,RdLo + Rm * Rs)	
	signed 16 * 16 bit	5E	SMULxy{cond} Rd, Rm, Rs			Rd := Rm[x] * Rs[y]	No shift/rotate.
	signed 32 * 16 bit	5E	SMULWy{cond} Rd, Rm, Rs			Rd := (Rm * Rs[y])[47:16]	No shift/rotate.
	signed accumulate 16 * 16	5E	SMLAxy{cond} Rd, Rm, Rs, Rn		Q	Rd := Rn + Rm[x] * Rs[y]	No shift/rotate.
	signed accumulate 32 * 16	5E	SMLAWy{cond} Rd, Rm, Rs, Rn		Q	Rd := Rn + (Rm * Rs[y])[47:16]	No shift/rotate.
	signed accumulate long 16 * 16	5E	SMLALxy{cond} RdLo, RdHi, Rm, Rs			RdHi,RdLo := RdHi,RdLo + Rm[x] * Rs[y]	No shift/rotate.
	Count leading zeroes	5	CLZ{cond} Rd, Rm			Rd := number of leading zeroes in Rm	
Logical	Test		TST{cond} Rn, <Oprnd2>	N Z C		Update CPSR flags on Rn AND Oprnd2	
	Test equivalence		TEQ{cond} Rn, <Oprnd2>	N Z C		Update CPSR flags on Rn EOR Oprnd2	
	AND		AND{cond}{S} Rd, Rn, <Oprnd2>	N Z C		Rd := Rn AND Oprnd2	
	EOR		EOR{cond}{S} Rd, Rn, <Oprnd2>	N Z C		Rd := Rn EOR Oprnd2	
	ORR		ORR{cond}{S} Rd, Rn, <Oprnd2>	N Z C		Rd := Rn OR Oprnd2	
	Bit Clear		BIC{cond}{S} Rd, Rn, <Oprnd2>	N Z C		Rd := Rn AND NOT Oprnd2	
	No operation		NOP			R0 := R0	Flags not affected.
	Shift/Rotate						See Table **Operand 2.**
Compare	Compare		CMP{cond} Rn, <Oprnd2>	N Z C V		Update CPSR flags on Rn - Oprnd2	
	negative		CMN{cond} Rn, <Oprnd2>	N Z C V		Update CPSR flags on Rn + Oprnd2	

● ARM命令セット2

Operation		§	Assembler	Action	Notes
Branch	Branch		B{cond} label	R15 := label	label must be within ±32Mb of current instruction.
	with link		BL{cond} label	R14 := R15-4, R15 := label	label must be within ±32Mb of current instruction.
	and exchange	4T	BX{cond} Rm	R15 := Rm, Change to Thumb if Rm[0] is 1	
	with link and exchange (1)	5T	BLX label	R14 := R15 - 4, R15 := label, Change to Thumb	Cannot be conditional. label must be within ±32Mb of current instruction.
	with link and exchange (2)	5T	BLX{cond} Rm	R14 := R15 - 4, R15 := Rm[31:1] Change to Thumb if Rm[0] is 1	
Load	Word		LDR{cond} Rd, <a_mode2>	Rd := [address]	
	User mode privilege		LDR{cond}T Rd, <a_mode2P>		
	branch (and exchange)		LDR{cond} R15, <a_mode2>	R15 := [address][31:1] (§ 5T: Change to Thumb if [address][0] is 1)	
	Byte		LDR{cond}B Rd, <a_mode2>	Rd := ZeroExtend[byte from address]	
	User mode privilege		LDR{cond}BT Rd, <a_mode2P>		
	signed	4	LDR{cond}SB Rd, <a_mode3>	Rd := SignExtend[byte from address]	
	Halfword	4	LDR{cond}H Rd, <a_mode3>	Rd := ZeroExtent[halfword from address]	
	signed	4	LDR{cond}SH Rd, <a_mode3>	Rd := SignExtend[halfword from address]	
Load multiple	Pop, or Block data load		LDM{cond}<a_mode4L> Rd{!}, <reglist-pc>	Load list of registers from [Rd]	
	return (and exchange)		LDM{cond}<a_mode4L> Rd{!}, <reglist+pc>	Load registers, R15 := [address][31:1] (§ 5T: Change to Thumb if [address][0] is 1)	
	and restore CPSR		LDM{cond}<a_mode4L> Rd{!}, <reglist+pc>^	Load registers, branch (§ 5T: and exchange), CPSR := SPSR	Use from exception modes only.
	User mode registers		LDM{cond}<a_mode4L> Rd, <reglist-pc>^	Load list of User mode registers from [Rd]	Use from privileged modes only.
Store	Word		STR{cond} Rd, <a_mode2>	[address] := Rd	
	User mode privilege		STR{cond}T Rd, <a_mode2P>	[address] := Rd	
	Byte		STR{cond}B Rd, <a_mode2>	[address][7:0] := Rd[7:0]	
	User mode privilege		STR{cond}BT Rd, <a_mode2P>	[address][7:0] := Rd[7:0]	
	Halfword	4	STR{cond}H Rd, <a_mode3>	[address][15:0] := Rd[15:0]	
Store multiple	Push, or Block data store		STM{cond}<a_mode4S> Rd{!}, <reglist>	Store list of registers to [Rd]	
	User mode registers		STM{cond}<a_mode4S> Rd{!}, <reglist>^	Store list of User mode registers to [Rd]	Use from privileged modes only.
Swap	Word	3	SWP{cond} Rd, Rm, [Rn]	temp := [Rn], [Rn] := Rm, Rd := temp	
	Byte	3	SWP{cond}B Rd, Rm, [Rn]	temp := ZeroExtend([Rn][7:0]), [Rn][7:0] := Rm[7:0], Rd := temp	
Coprocessors	Data operations	2	CDP{cond} p<cpnum>, <op1>, CRd, CRn, CRm, <op2>	Coprocessor defined	
		5	CDP2 p<cpnum>, <op1>, CRd, CRn, CRm, <op2>		Cannot be conditional.
	Move to ARM reg from coproc	2	MRC{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2>		
		5	MRC2 p<cpnum>, <op1>, Rd, CRn, CRm, <op2>		Cannot be conditional.
	Move to coproc from ARM reg	2	MCR{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2>		
		5	MCR2 p<cpnum>, <op1>, Rd, CRn, CRm, <op2>		Cannot be conditional.
	Load	2	LDC{cond} p<cpnum>, CRd, <a_mode5>		
		5	LDC2 p<cpnum>, CRd, <a_mode5>		Cannot be conditional.
	Store	2	STC{cond} p<cpnum>, CRd, <a_mode5>		
		5	STC2 p<cpnum>, CRd, <a_mode5>		Cannot be conditional.
Software interrupt			SWI{cond} <immed_24>	Software interrupt processor exception	24-bit value encoded in instruction.
Breakpoint		5	BKPT <immed_16>	Prefetch abort *or* enter debug state	Cannot be conditional.

● Thumb命令セット1

All Thumb registers are Lo (R0-R7) except where specified. Hi registers are R8-R15.

Operation		§	Assembler	Update flags	Action	Notes
Move	Immediate		MOV Rd, #<immed_8>	✓	Rd := immed_8	8-bit immediate value.
	Lo to Lo		MOV Rd, Rm	✓	Rd := Rm	
	Hi to Lo, Lo to Hi, Hi to Hi		MOV Rd, Rm	×	Rd := Rm	Not Lo to Lo
Arithmetic	Add		ADD Rd, Rn, #<immed_3>	✓	Rd := Rn + immed_3	3-bit immediate value.
	Lo and Lo		ADD Rd, Rn, Rm	✓	Rd := Rn + Rm	
	Hi to Lo, Lo to Hi, Hi to Hi		ADD Rd, Rm	×	Rd := Rd + Rm	Not Lo to Lo
	immediate		ADD Rd, #<immed_8>	✓	Rd := Rd + immed_8	8-bit immediate value.
	with carry		ADC Rd, Rm	✓	Rd := Rd + Rm + C-bit	
	value to SP		ADD SP, #<immed_7*4>	×	SP := SP + immed_7 * 4	9-bit immediate value (word-aligned).
	form address from SP		ADD Rd, SP, #<immed_8*4>	×	Rd := SP + immed_8 * 4	10-bit immediate value (word-aligned).
	form address from PC		ADD Rd, PC, #<immed_8*4>	×	Rd := (PC AND 0xFFFFFFFC) + immed_8 * 4	10-bit immediate value (word-aligned).
	Subtract		SUB Rd, Rn, Rm	✓	Rd := Rn - Rm	
	immediate 3		SUB Rd, Rn, #<immed_3>	✓	Rd := Rn - immed_3	3-bit immediate value.
	immediate 8		SUB Rd, #<immed_8>	✓	Rd := Rd - immed_8	8-bit immediate value.
	with carry		SBC Rd, Rm	✓	Rd := Rd - Rm - NOT C-bit	
	value from SP		SUB SP, #<immed_7*4>	×	SP := SP - immed_7 * 4	9-bit immediate value (word-aligned).
	Negate		NEG Rd, Rm	✓	Rd := - Rm	
	Multiply		MUL Rd, Rm	✓	Rd := Rm * Rd	
	Compare		CMP Rn, Rm	✓	update CPSR flags on Rn - Rm	Can be Lo to Lo, Lo to Hi, Hi to Lo, or Hi to Hi.
	negative		CMN Rn, Rm	✓	update CPSR flags on Rn + Rm	
	immediate		CMP Rn, #<immed_8>	✓	update CPSR flags on Rn - immed_8	8-bit immediate value.
	No operation		NOP	×	R8 := R8	Flags not affected.
Logical	AND		AND Rd, Rm	✓	Rd := Rd AND Rm	
	Exclusive OR		EOR Rd, Rm	✓	Rd := Rd EOR Rm	
	OR		ORR Rd, Rm	✓	Rd := Rd OR Rm	
	Bit clear		BIC Rd, Rm	✓	Rd := Rd AND NOT Rm	
	Move NOT		MVN Rd, Rm	✓	Rd := NOT Rm	
	Test bits		TST Rn, Rm	✓	update CPSR flags on Rn AND Rm	
Shift/rotate	Logical shift left		LSL Rd, Rm, #<immed_5>	✓	Rd := Rm << immed_5	5-bit immediate shift. Allowed shifts 0-31.
			LSL Rd, Rs	✓	Rd := Rd << Rs	
	Logical shift right		LSR Rd, Rm, #<immed_5>	✓	Rd := Rm >> immed_5	5-bit immediate shift. Allowed shifts 1-32.
			LSR Rd, Rs	✓	Rd := Rd >> Rs	
	Arithmetic shift right		ASR Rd, Rm, #<immed_5>	✓	Rd := Rm ASR immed_5	5-bit immediate shift. Allowed shifts 1-32.
			ASR Rd, Rs	✓	Rd := Rd ASR Rs	
	Rotate right		ROR Rd, Rs	✓	Rd := Rd ROR Rs	
Branch	Conditional branch		B{cond} label		R15 := label	label must be within -252 to +258 bytes of current instruction. See Table Condition Field (ARM side). AL not allowed.
	Unconditional branch		B label		R15 := label	label must be within ±2Kb of current instruction.
	Long branch with link		BL label		R14 := R15 - 2, R15 := label	Encoded as two Thumb instructions. label must be within ±4Mb of current instruction.
	Branch and exchange		BX Rm		R15 := Rm AND 0xFFFFFFFE	Change to ARM state if Rm[0] = 0.
	Branch with link and exchange	5T	BLX label		R14 := R15 - 2, R15 := label Change to ARM	Encoded as two Thumb instructions. label must be within ±4Mb of current instruction.
	Branch with link and exchange	5T	BLX Rm		R14 := R15 - 2, R15 := Rm AND 0xFFFFFFFE Change to ARM if Rm[0] = 0	
Software Interrupt			SWI <immed_8>		Software interrupt processor exception	8-bit immediate value encoded in instruction.
Breakpoint		5T	BKPT <immed_8>		Prefetch abort *or* enter debug state	

● Thumb命令セット2

Operation		§	Assembler	Action	Notes
Load	with immediate offset, word		LDR Rd, [Rn, #<immed_5*4>]	Rd := [Rn + immed_5 * 4]	
	halfword		LDRH Rd, [Rn, #<immed_5*2>]	Rd := ZeroExtend([Rn + immed_5 * 2][15:0])	Clears bits 31:16
	byte		LDRB Rd, [Rn, #<immed_5>]	Rd := ZeroExtend([Rn + immed_5][7:0])	Clears bits 31:8
	with register offset, word		LDR Rd, [Rn, Rm]	Rd := [Rn + Rm]	
	halfword		LDRH Rd, [Rn, Rm]	Rd := ZeroExtend([Rn + Rm][15:0])	Clears bits 31:16
	signed halfword		LDRSH Rd, [Rn, Rm]	Rd := SignExtend([Rn + Rm][15:0])	Sets bits 31:16 to bit 15
	byte		LDRB Rd, [Rn, Rm]	Rd := ZeroExtend([Rn + Rm][7:0])	Clears bits 31:8
	signed byte		LDRSB Rd, [Rn, Rm]	Rd := SignExtend([Rn + Rm][7:0])	Sets bits 31:8 to bit 7
	PC-relative		LDR Rd, [PC, #<immed_8*4>]	Rd := [(PC AND 0xFFFFFFFC) + immed_8 * 4]	
	SP-relative		LDR Rd, [SP, #<immed_8*4>]	Rd := [SP + immed_8 * 4]	
	Multiple		LDMIA Rn!, <reglist>	Loads list of registers	Always updates base register.
Store	with immediate offset, word		STR Rd, [Rn, #<immed_5*4>]	[Rn + immed_5 * 4] := Rd	
	halfword		STRH Rd, [Rn, #<immed_5*2>]	[Rn + immed_5 * 2][15:0] := Rd[15:0]	Ignores Rd[31:16]
	byte		STRB Rd, [Rn, #<immed_5>]	[Rn + immed_5][7:0] := Rd[7:0]	Ignores Rd[31:8]
	with register offset, word		STR Rd, [Rn, Rm]	[Rn + Rm] := Rd	
	halfword		STRH Rd, [Rn, Rm]	[Rn + Rm][15:0] := Rd[15:0]	Ignores Rd[31:16]
	byte		STRB Rd, [Rn, Rm]	[Rn + Rm][7:0] := Rd[7:0]	Ignores Rd[31:8]
	SP-relative, word		STR Rd, [SP, #<immed_8*4>]	[SP + immed_8 * 4] := Rd	
	Multiple		STMIA Rn!, <reglist>	Stores list of registers	Always updates base register.
Push/ Pop	Push		PUSH <reglist>	Push registers onto stack	Full descending stack.
	Push with link		PUSH <reglist, LR>	Push LR and registers onto stack	
	Pop		POP <reglist>	Pop registers from stack	
	Pop and return		POP <reglist, PC>	Pop registers, branch to address loaded to PC	
	Pop and return with exchange	5T	POP <reglist, PC>	Pop, branch, and change to ARM state if address[0] = 0	

Proprietary Notice

Document Number

ARM QRC 0001D

Change Log

Issue	Date	By	Change
A	June 1995	BJH	First Release
B	Sept 1996	BJH	Second Release
C	Nov 1998	BJH	Third Release
D	Oct 1999	CKS	Fourth Release

● ベクタ浮動小数点命令セット

Key to Tables	
{cond}	See Table **Condition Field** (on ARM side).
<S/D>	S (single precision) or D (double precision).
<S/D/X>	As above, or X (unspecified precision).
Fd, Fn, Fm	Sd, Sn, Sm (single precision), or Dd, Dn, Dm (double precision).

{E}	E : raise exception on any NaN. Without E : raise exception only on signaling NaNs.
{Z}	Round towards zero. Overrides FPSCR rounding mode.
<VFPregs>	A comma separated list of *consecutive* VFP registers, enclosed in braces ({ and }).
<VFPsysreg>	FPSCR, or FPSID.

Operation		Assembler	Exceptions	Action	Notes
Vector arithmetic	Multiply	FMUL<S/D>{cond} Fd, Fn, Fm	IO, OF, UF, IX	Fd := Fn * Fm	
	negative	FNMUL<S/D>{cond} Fd, Fn, Fm	IO, OF, UF, IX	Fd := - (Fn * Fm)	
	accumulate	FMAC<S/D>{cond} Fd, Fn, Fm	IO, OF, UF, IX	Fd := Fd + (Fn * Fm)	
	deduct	FNMAC<S/D>{cond} Fd, Fn, Fm	IO, OF, UF, IX	Fd := Fd - (Fn * Fm)	
	negate and accumulate	FMSC<S/D>{cond} Fd, Fn, Fm	IO, OF, UF, IX	Fd := -Fd + (Fn * Fm)	
	negate and deduct	FNMSC<S/D>{cond} Fd, Fn, Fm	IO, OF, UF, IX	Fd := -Fd - (Fn * Fm)	
	Add	FADD<S/D>{cond} Fd, Fn, Fm	IO, OF, IX	Fd := Fn + Fm	
	Subtract	FSUB<S/D>{cond} Fd, Fn, Fm	IO, OF, IX	Fd := Fn - Fm	
	Divide	FDIV<S/D>{cond} Fd, Fn, Fm	IO, DZ, OF, UF, IX	Fd := Fn / Fm	
	Copy	FCPY<S/D>{cond} Fd, Fm		Fd := Fm	
	Absolute	FABS<S/D>{cond} Fd, Fm		Fd := abs(Fm)	
	Negative	FNEG<S/D>{cond} Fd, Fm		Fd := -Fm	
	Square root	FSQRT<S/D>{cond} Fd, Fm	IO, IX	Fd := sqrt(Fm)	
Scalar compare		FCMP{E}<S/D>{cond} Fd, Fm	IO	Set FPSCR flags on Fd - Fm	Use FMSTAT to transfer flags.
	Compare with zero	FCMP{E}Z<S/D>{cond} Fd	IO	Set FPSCR flags on Fd - 0	Use FMSTAT to transfer flags.
Scalar convert	Single to double	FCVTDS{cond} Dd, Sm	IO	Dd := convertStoD(Sm)	
	Double to single	FCVTSD{cond} Sd, Dm	IO, OF, UF, IX	Sd := convertDtoS(Dm)	
	Unsigned integer to float	FUITO<S/D>{cond} Fd, Sm		Fd := convertUItoF(Sm)	
	Signed integer to float	FSITO<S/D>{cond} Fd, Sm	IX	Fd := convertSItoF(Sm)	
	Float to unsigned integer	FTOUI{Z}<S/D>{cond} Sd, Fm	IO, IX	Sd := convertFtoUI(Fm)	
	Float to signed integer	FTOSI{Z}<S/D>{cond} Sd, Fm	IO, IX	Sd := convertFtoSI(Fm)	
Save VFP registers		FST<S/D>{cond} Fd, [Rn{, #<immed_8*4>}]		[address] := Fd	
	Multiple, unindexed	FSTMIA<S/D/X>{cond} Rn, <VFPregs>		Saves list of VFP registers, starting at address in Rn.	
	increment after	FSTMIA<S/D/X>{cond} Rn!, <VFPregs>		synonym: FSTMEA (empty ascending)	
	decrement before	FSTMDB<S/D/X>{cond} Rn!, <VFPregs>		synonym: FSTMFD (full descending)	
Load VFP registers		FLD<S/D>{cond} Fd, [Rn{, #<immed_8*4>}]		Fd := [address]	
	Multiple, unindexed	FLDMIA<S/D/X>{cond} Rn, <VFPregs>		Loads list of VFP registers, starting at address in Rn.	
	increment after	FLDMIA<S/D/X>{cond} Rn!, <VFPregs>		synonym: FLDMFD (full descending)	
	decrement before	FLDMDB<S/D/X>{cond} Rn!, <VFPregs>		synonym: FLDMEA (empty ascending)	
Transfer registers	ARM to single	FMSR{cond} Sn, Rd		Sn := Rd	
	Single to ARM	FMRS{cond} Rd, Sn		Rd := Sn	
	ARM to lower half of double	FMDLR{cond} Dn, Rd		Dn[31:0] := Rd	Use with FMDHR.
	Lower half of double to ARM	FMRDL{cond} Rd, Dn		Rd := Dn[31:0]	Use with FMRDH.
	ARM to upper half of double	FMDHR{cond} Dn, Rd		Dn[63:32] := Rd	Use with FMDLR.
	Upper half of double to ARM	FMRDH{cond} Rd, Dn		Rd := Dn[63:32]	Use with FMRDL.
	ARM to VFP system register	FMXR{cond} <VFPsysreg>, Rd		VFPsysreg := Rd	Stalls ARM until all VFP ops complete.
	VFP system register to ARM	FMRX{cond} Rd, <VFPsysreg>		Rd := VFPsysreg	Stalls ARM until all VFP ops complete.
	FPSCR flags to CPSR	FMSTAT{cond}		CPSR flags := FPSCR flags	Equivalent to FMRX R15, FPSCR

Exceptions	
IO	Invalid operation
OF	Overflow
UF	Underflow
IX	Inexact result
DZ	Division by zero

FPSCR format								Rounding		(Stride - 1)*3			Vector length - 1						Exception trap enable bits								Cumulative exception bits				
31	30	29	28				24	23	22	21	20		18	17	16				12	11	10	9	8				4	3	2	1	0
N	Z	C	V				FZ	RMODE		STRIDE			LEN						IXE	UFE	OFE	DZE	IOE				IXC	UFC	OFC	DZC	IOC

FZ: 1 = flush to zero mode. Rounding: 0 = round to nearest, 1 = towards +∞, 2 = towards -∞, 3 = towards zero. (Vector length * Stride) must not exceed 4 for double precision operands.

If Fd is S0-S7 or D0-D3, operation is Scalar (regardless of vector length).	If Fd is S8-S31 or D4-D15, and Fm is S0-S7 or D0-D3, operation is Mixed (Fm scalar, others vector).
If Fd is S8-S31 or D4-D15, and Fm is S8-S31 or D4-D15, operation is Vector.	S0-S7 (or D0-D3), S8-S15 (D4-D7), S16-S23 (D8-D11), S24-S31 (D12-D15) each form a circulating bank of registers.

● ARMアドレシング・モード

Addressing Mode 2 - Word and Unsigned Byte Data Transfer

Pre-indexed	Immediate offset	[Rn, #+/-<immed_12>]{!}	
	Zero offset	[Rn]	Equivalent to [Rn,#0]
	Register offset	[Rn, +/-Rm]{!}	
	Scaled register offset	[Rn, +/-Rm, LSL #<immed_5>]{!}	Allowed shifts 0-31
		[Rn, +/-Rm, LSR #<immed_5>]{!}	Allowed shifts 1-32
		[Rn, +/-Rm, ASR #<immed_5>]{!}	Allowed shifts 1-32
		[Rn, +/-Rm, ROR #<immed_5>]{!}	Allowed shifts 1-31
		[Rn, +/-Rm, RRX]{!}	
Post-indexed	Immediate offset	[Rn], #+/-<immed_12>	
	Register offset	[Rn], +/-Rm	
	Scaled register offset	[Rn], +/-Rm, LSL #<immed_5>	Allowed shifts 0-31
		[Rn], +/-Rm, LSR #<immed_5>	Allowed shifts 1-32
		[Rn], +/-Rm, ASR #<immed_5>	Allowed shifts 1-32
		[Rn], +/-Rm, ROR #<immed_5>	Allowed shifts 1-31
		[Rn], +/-Rm, RRX	

Addressing Mode 2 (Post-indexed only)

Post-indexed	Immediate offset	[Rn], #+/-<immed_12>	
	Zero offset	[Rn]	Equivalent to [Rn],#0
	Register offset	[Rn], +/-Rm	
	Scaled register offset	[Rn], +/-Rm, LSL #<immed_5>	Allowed shifts 0-31
		[Rn], +/-Rm, LSR #<immed_5>	Allowed shifts 1-32
		[Rn], +/-Rm, ASR #<immed_5>	Allowed shifts 1-32
		[Rn], +/-Rm, ROR #<immed_5>	Allowed shifts 1-31
		[Rn], +/-Rm, RRX	

Addressing Mode 3 - Halfword and Signed Byte Data Transfer

Pre-indexed	Immediate offset	[Rn, #+/-<immed_8>]{!}	
	Zero offset	[Rn]	Equivalent to [Rn,#0]
	Register	[Rn, +/-Rm]{!}	
Post-indexed	Immediate offset	[Rn], #+/-<immed_8>	
	Register	[Rn], +/-Rm	

Addressing Mode 4 - Multiple Data Transfer

Block load		Stack pop	
IA	Increment After	FD	Full Descending
IB	Increment Before	ED	Empty Descending
DA	Decrement After	FA	Full Ascending
DB	Decrement Before	EA	Empty Ascending
Block store		**Stack push**	
IA	Increment After	EA	Empty Ascending
IB	Increment Before	FA	Full Ascending
DA	Decrement After	ED	Empty Descending
DB	Decrement Before	FD	Full Descending

Addressing Mode 5 - Coprocessor Data Transfer

Pre-indexed	Immediate offset	[Rn, #+/-<immed_8*4>]{!}	
	Zero offset	[Rn]	Equivalent to [Rn,#0]
Post-indexed	Immediate offset	[Rn], #+/-<immed_8*4>	
Unindexed	No offset	[Rn], {8-bit copro. option}	

ARM architecture versions

n	ARM architecture version *n* and above.
*n*T	T variants of ARM architecture version *n* and above.
M	ARM architecture version 3M, and 4 and above excluding xM variants
*n*E	E variants of ARM architecture version *n* and above.

Operand 2

Immediate value	#<immed_8r>	
Logical shift left immediate	Rm, LSL #<immed_5>	Allowed shifts 0-31
Logical shift right immediate	Rm, LSR #<immed_5>	Allowed shifts 1-32
Arithmetic shift right immediate	Rm, ASR #<immed_5>	Allowed shifts 1-32
Rotate right immediate	Rm, ROR #<immed_5>	Allowed shifts 1-31
Register	Rm	
Rotate right extended	Rm, RRX	
Logical shift left register	Rm, LSL Rs	
Logical shift right register	Rm, LSR Rs	
Arithmetic shift right register	Rm, ASR Rs	
Rotate right register	Rm, ROR Rs	

PSR fields (use at least one suffix)

Suffix	Meaning	
c	Control field mask byte	PSR[7:0]
f	Flags field mask byte	PSR[31:24]
s	Status field mask byte	PSR[23:16]
x	Extension field mask byte	PSR[15:8]

Condition Field {cond}

Mnemonic	Description	Description (VFP)
EQ	Equal	Equal
NE	Not equal	Not equal, or unordered
CS / HS	Carry Set / Unsigned higher or same	Greater than or equal, or unordered
CC / LO	Carry Clear / Unsigned lower	Less than
MI	Negative	Less than
PL	Positive or zero	Greater than or equal, or unordered
VS	Overflow	Unordered (at least one NaN operand)
VC	No overflow	Not unordered
HI	Unsigned higher	Greater than, or unordered
LS	Unsigned lower or same	Less than or equal
GE	Signed greater than or equal	Greater than or equal
LT	Signed less than	Less than, or unordered
GT	Signed greater than	Greater than
LE	Signed less than or equal	Less than or equal, or unordered
AL	Always (normally omitted)	Always (normally omitted)

Key to tables

{!}	Updates base register after data transfer if ! present. (Post-indexed always updates.)
<immed_8r>	A 32-bit constant, formed by right-rotating an 8-bit value by an even number of bits.
+/-	+ or -. (+ may be omitted.)

索　　引

■た行■

■な行■

■は行■

■ま行■

■や行■

■ら行■

■わ行■

■英字■

■数字■

<< 付属CD-ROMについて >>

本書の付属CD-ROMには，ARM Limited 提供によるWindows版のARMソフトウェア開発ツール「ARM Developer Suite」（評価版）を収録しています．**評価版**ですので，使用期間は「45日間」に限定されています．このツールに関する技術サポートはありません．また，使用方法，使用上の注意などのドキュメント類はすべて英語のみです．あらかじめご了承ください．

収録しているツールは次のとおりです．

■ARM Developer Suite Evaluation Version1.1

ARM命令セット，Thumb命令セットをサポートするクロス開発環境．動作環境はWindows95/98/2000/NT4.0.

(1) コード生成ツール

ARM命令，Thumb命令をサポートするCおよび Embedded C++ (EC++)，アセンブラ，リンカが含まれます．

(2) 統合開発環境(IDE)

統合開発環境としては，CodeWarrior (Metrowerks) を採用しています．

(3) デバッガ

評価版用デバッガとしてARM eXtended Debugger (AXD) が含まれます．

(4) 命令セット・シミュレータ

ARMおよびThumbに対応した命令セット・シミュレータARMulatorが用意されています．

(5) ドキュメント

オンライン・ドキュメントがPDFファイルで含まれています．

<<動作環境>>

(1) Pentium以上のCPUを搭載するIBM PC互換機
(2) 32Mバイト以上のRAM
(3) 必要なハード・ディスク容量は，インストールの仕方により変化する．すべてをインストールする場合は180Mバイト以上必要．
(4) CD-ROMドライブ
(5) OSはWindows95/98/2000/NT4.0
(6) オンライン・マニュアルを見るためのAdobe社のAcrobat Reader.

注意：付属CD-ROMには英語版Acrobat Readerが収録されています．日本語対応版ではありませんので，すでに日本語版Acrobat Readerをご使用の方は，本付属CD-ROMで再度Acrobat Readerをインストールされないようにご注意ください．英語版で上書きされてしまいます．

ご使用に当たっては，CD-ROM上のREADME.txtファイルおよび各ドキュメントを十分にお読みください．

改訂　ARMプロセッサ　　CD-ROM付き

2001年12月18日　初版発行

著　者　Steve Furber
監訳者　アーム株式会社
発行人　蒲　生　良　治
発行所　CQ出版株式会社
〒170-8461　東京都豊島区巣鴨1-14-2
出版部　☎03-5395-2126
販売部　☎03-5395-2141
振　替　00100-7-10665

(定価はカバーに表示してあります)
Printed in Japan

DTP・印刷・製本　クニメディア株式会社
乱丁・落丁本はお取り替えいたします

ISBN4-7898-3357-7

C3055 ¥3400E

CQ出版社

定価：本体3,400円（税別）

9784789833578

1923055034006

改訂ARMプロセッサ

32ビットRISCのシステム・アーキテクチャ

